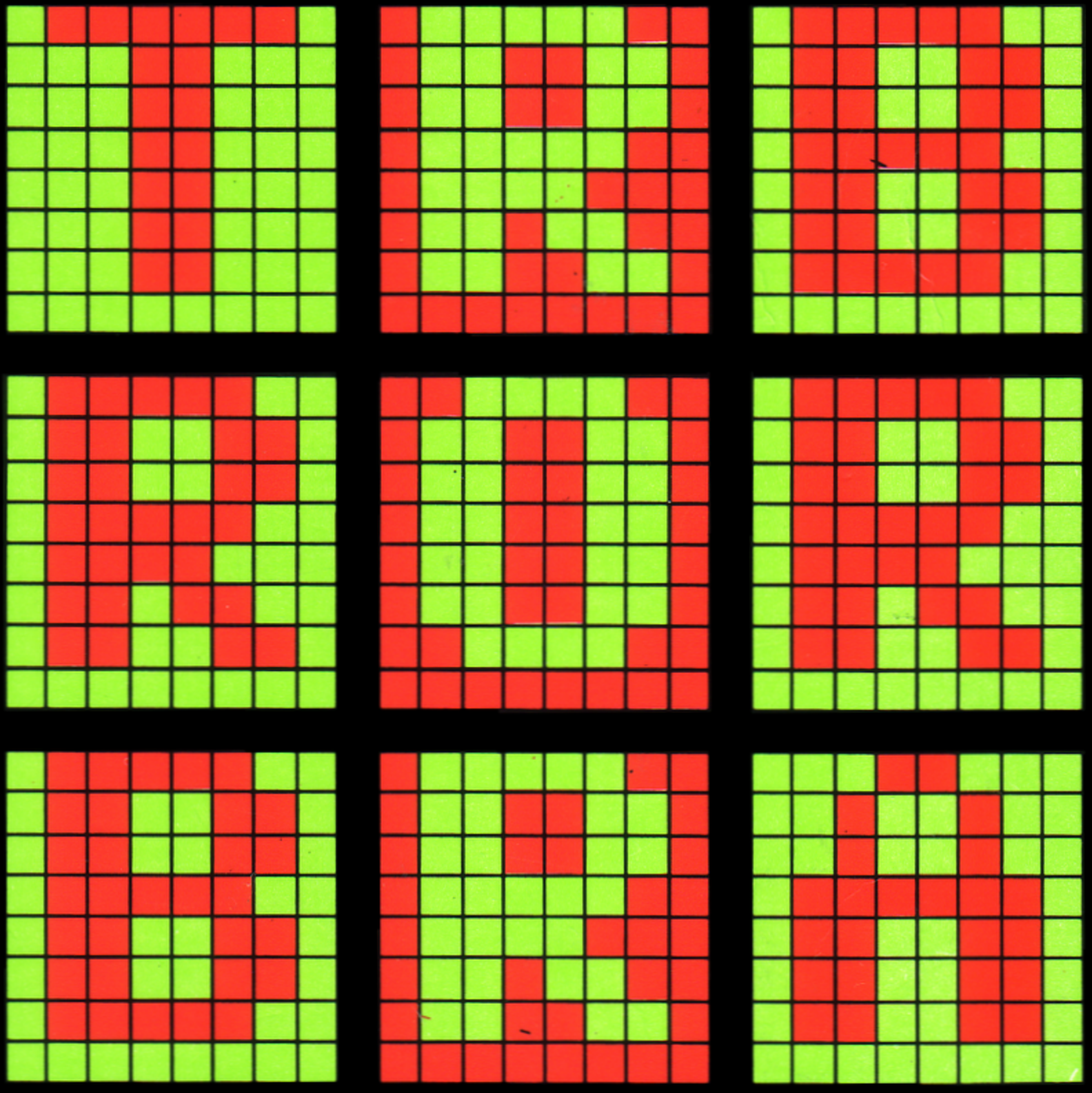
THE NEW ADVANCED USER GUIDE

for BBC Master, Compact, B, B+ & Electron



Dickens & Holmes

ADDER

1	Introduction to machine code	3
2	The BASIC Assembler	6
3	Machine Code Arithmetic	15
4	Addressing Modes	19
5	The 6502 Instruction Set	28
6	Introduction to the OS	100
7	Events	119
8	Interrupts	124
9	Buffers control and management	136
10	Escape related calls	147
11	An Introduction to Hardware	150
12	Memory	155
13	Video/Graphics System	165
14	Keyboard routines	218
15	Serial I/O (RS232/423)	233
16	Filing System Implementations	250
17	Paged ROMs	283
18	Second processors	327
19	Clocks	352
20	ADC system	365
21	Sound and speech systems	372
22	User/printer and system VIAs	380
23	The 1MHz bus & cartridge i/f	402
24	Miscellaneous topics	419
	Glossary	430
	Bibliography	433

OS call	address	vector	address
		USERV	&200
		BRKV	&202
		1RQ1V	&204
		1RQ2V	&206
OSCLI	&FFF7	CLIV	&208
OSBYTE	&FFF4	BYTEV	&20A
OSWORD	&FFF1	WORDV	&20C
OSWRCH	&FFEE	WRCHV	&20E
OSNEWL	&FFE7	-	-
OSASCI	&FFE3	-	-
OSRDCH	&FFE0	RDCHV	&210
OSFILE	&FFDD	FILEV	&212
OSARGS	&FFDA	ARGSV	&214
OSBGET	&FFD7	BGETV	&216
OSBPUT	&FFD4	BPUTV	&218
OSGBPB	&FFD1	GBPBV	&21A
OSFIND	&FFCE	FINDV	&21C
		FSCV	&21E
		EVNTV	&220
		UPTV	&222
		NETV	&224
		VDUV	&226
		KEYV	&228
		INSV	&22A
		REMV	&22C
		CNPV	&22E
NVWRCH	&FFCB	-	-
NVRDCH	&FFC8	-	-
GSREAD	&FFC5	-	-
GSINIT	&FFC2	-	-
OSEVEN	&FFBF	-	-
OSRDSC	&FFB9	-	-
OSWRSC	&FFB3	-	-

THE NEW ADVANCED USER GUIDE

BBC Master, Compact, B, B+ & Electron

Holmes & Dickens

ADDER

THE NEW ADVANCED USER GUIDE

for BBC Master, Compact, B, B+ & Electron

This is the authoritative guide to the Acorn-BBC range of computers including the Master 128, Master Compact, BBC model B, B Plus and Electron. Following the original best-selling *Advanced User Guide for the BBC Micro*, this thoroughly revised and updated book provides users with an invaluable reference guide covering all aspects of the system hardware and software including:

- The BASIC assembler
- A full 65C02 reference section
- ALL the *FX/OSBYTE calls
- Implementing paged ROM software
- Using events, interrupts & buffers
- Programming the video circuitry
- Shadow RAM control
- Video and serial ULAs described
- Analogue port and Master Compact analogue emulator
- Using the Tube™ and second processors
- Master CMOS RAM/Real Time Clock
- Master Compact EEPROM
- Programming the 1770 disc controller

Example programs include:
- Expanded printer buffer paged ROM
- Rapid data transfer across the Tube™ paged ROM
- Using the real time clock alarm interrupt
- Reading IBM PC format discs using the 1770 disc controller

Hardware examples include:
- Constructing and using a light pen
- Using the cartridge interfaces
- Interfacing with the 1MHz bus and User port

Adrian Dickens and Mark Holmes are both respected authorities on the BBC Micro and between them have extensive experience of using, programming and writing about the BBC Micro in all its forms. The book is liberally sprinkled with examples and ideas for extracting the most from this range of computers. No serious programmer of the Acorn-BBC series machines can consider not owning a copy of this reference manual.

ISBN 0 947929 05 3

£19.95

The New Advanced User Guide

for the Master, Master Compact, BBC model B, B+ & Electron.

by

Mark Holmes & Adrian Dickens

ADDER

Published in the United Kingdom by Adder Publishing Limited, Cambridge

Published in the United Kingdom by:

Adder Publishing Limited,
P.O.Box 148,
Cambridge CB1 2EQ

ISBN 0 947 929 05 3

Copyright ©1987 Mark Holmes & Adrian Dickens

First published September 1987

The authors would like to thank Nigel Dickens for his assistance in the production of this book.

All rights reserved. This book is copyright. No part of this book may be copied or stored by any means whatsoever whether mechanical, photographic or electronic, except for private or study use as defined in the Copyright Act. All enquiries should be addressed to the publishers. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of information contained herein.

Please note that within this text the words Acorn™, Tube™ and Econet™ are registered trademarks of Acorn Computers Limited. CP/M™ is a registered trademark of Digital Research Inc. All references in this book to the *BBC Micro, Master 128* and *Master Compact* refer to the computers manufactured by Acorn Computers Limited under license from the British Broadcasting Corporation.

This book was prepared using Microsoft Word (3.0) and MacDraw on an Apple Macintosh Plus computer. Programs were written on BBC Model B, B+ and Master 128 computers and the listings transferred to the Macintosh using a serial link. The book was typeset on an Apple Laserwriter Plus to produce A4 sized masters which were reduced to A5 by the printers to achieve an effective resolution of 424 dpi.

Printed in Great Britain by Burlington Press Ltd., Foxton, Cambridge.

Dedicated to the delectable ***Sarah Jane***

Contents

1 Introduction for those new to machine code 3
2 The BASIC Assembler 6
2.1 The assembler 7
2.2 OPT, assembler 8
2.3 The Location Counter P% 9
2.4 Labels 10
2.5 Forward Referencing and Two Pass Assembly 10
2.6 The EQUate Facility in Level 2 BASIC 10
2.7 Handling errors with BRK 12
2.8 Entering machine code from BASIC - CALL 12
2.9 Conditional Assembly and Macros 13
2.10 User Zero Page 14
3 Machine Code Arithmetic 15
3.1 2's Complement 15
3.2 Binary Coded Decimal 17
3.3 BCD and the 65C12 18
4 Addressing Modes 19
4.1 Implicit addressing 19
4.2 Accumulator addressing 19
4.3 Immediate addressing 19
4.4 Absolute addressing 20
4.5 Zero page 20
4.6 Indirect addressing 21
4.7 Absolute,X or Y addressing 22
4.8 Zero page,X addressing 22
4.9 Pre-indexed indirect addressing 23
4.10 Post-indexed indirect addressing 24
4.11 Relative addressing 25
4.12 Addressing modes and the 65C12 25
5 The 6502 Instruction Set 28
5.1 The 6502 registers and abbreviations 28
5.2 The Assembler Mnemonics 29
6 Introduction to the OS 100
6.1 Operating System Calls 102
6.2 I/O routines 103
6.3 OSBYTE 109
6.4 Filing System Calls 112
6.5 Miscellaneous OS calls 112
6.6 OS allocation and use of memory 114
7 Events 119
7.1 The event vector, EVNTV 119
7.2 Enable/disable event OSBYTE calls 121

7.3 OSEVEN 122
7.4 An example using events. 122
8 Interrupts **124**
8.1 Non Maskable Interrupts 125
8.2 Maskable Interrupts 125
8.3 The operating system interrupt handling routine 125
8.4 Serial system interrupts 126
8.5 System VIA interrupts 127
8.6 User VIA interrupts 129
8.7 Intercepting interrupts 129
8.8 Read/write User 6522 IRQ bit mask OSBYTE 130
8.9 Read/write 6850 IRQ bit mask OSBYTE 131
8.10 Read/write System 6522 IRQ bit mask OSBYTE 131
8.11 Read/write Electron ULA IRQ mask OSBYTE 132
8.12 BRK/error associated calls 132
8.13 The BRK vector &202 132
8.14 Read ROM no. active at last BRK OSBYTE call 135
9 Buffers control and management **136**
9.1 Insert value into buffer vector, INSV 136
9.2 Remove value from buffer vector 137
9.3 Count/purge buffer vector 137
9.4 Using the buffer vectors 138
9.5 Flush specific buffer OSBYTE 143
9.6 Flush selected buffer class OSBYTE call 143
9.7 Read buffer status OSBYTE 143
9.8 Insert value into buffer OSBYTE 144
9.9 Get character from buffer OSBYTE 145
9.10 Examine buffer status OSBYTE 145
9.11 Insert char. into i/p buffer OSBYTE call 146
10 Escape related calls **147**
10.1 Clear escape condition OSBYTE call 147
10.2 Set escape condition OSBYTE call 147
10.3 Clear escape + effects OSBYTE call 147
10.4 Read/write escape disable OSBYTE call 148
10.5 Read/write ESCAPE character OSBYTE call 148
10.6 Read/write ESCAPE key status OSBYTE call 149
10.7 Read/write ESCAPE effects OSBYTE call 149
11 An Introduction to Hardware **150**
12 Memory **155**
12.1 Memory map overview 155
12.2 OSBYTE calls concerning memory use 157
12.3 Paged ROM and RAM hardware control 161
12.4 RAM Access Control 161
13 Video/Graphics System **165**
13.1 O.S. Video Support 165

13.2 Video memory use 184
13.3 The Video hardware 187
13.4 Screen mode memory maps 210
14 Keyboard routines **218**
14.1 Key values 219
14.2 Read key with time limit OSBYTE call 221
14.3 Keyboard scan 221
14.4 Keyboard scan from &10 OSBYTE call 222
14.5 Write current keys pressed OSBYTE call 222
14.6 Read key translation table address OSBYTE call 223
14.7 Set keyboard auto-repeat delay 224
14.8 Set keyboard auto-repeat period 224
14.9 Function keys 225
14.10 Reflect keyboard status in keyboard LEDs OSBYTE 228
14.11 Read/write keyboard disable 228
14.12 Read/write keyboard status byte OSBYTE call 229
14.13 Read/write keyboard semaphore OSBYTE call 230
14.14 Set base for numeric keypad OSBYTE call 230
14.15 Read/write shift key effect OSBYTE call 230
14.16 Electron firm keys 231
14.17 Read/write TAB key character OSBYTE call 231
14.18 Read/write Escape character OSBYTE call 232
15 Serial I/O (RS232/423) **233**
15.1 The RS232C standard 233
15.2 The Acorn RS423/RS232 implementation 239
15.3 OS calls for using the serial port 240
16 Filing System Implementations **250**
16.1 Filing system calls 250
16.2 Master Series Filing Systems 259
16.3 The main filing systems 261
16.4 Floppy Disc Hardware 270
17 Paged ROMs **283**
17.1 Paged ROM header format 284
17.2 Paged ROM/RAM installation 287
17.3 Language ROMs 291
17.4 Service ROMs 294
17.5 Serially accessed ROMs & the *ROM filing system 313
17.6 Paged ROM associated routines 321
18 Second processors **327**
18.1 Tube system 32 bit addressing 327
18.2 OS calls made from second processors 328
18.3 The Tube ULA 328
18.4 The Tube software on the i/o processor 330
18.5 The 6502 as a typical second processor 331
18.6 Using OS calls and vectors 332

18.7 Memory allocation and usage 336
18.8 Protocol for transferring data across the Tube 338
18.9 Tube OSBYTE and OSWORD calls 347
18.10 The Z80 second processor 347
18.11 The 32016 second processor 350
19 Clocks **352**
19.1 System clock 352
19.2 Interval timer 352
19.3 Read timer state switch OSBYTE call 353
19.4 CMOS Real Time Clock 353
19.5 CMOS RAM/EEPROM 356
19.6 CMOS RAM/RTC hardware (Master only) 358
20 ADC system **365**
20.1 ADC operating system calls 365
20.2 ADC Hardware 369
21 Sound and speech systems **372**
21.1 Sound 372
21.2 The 76489 sound chip 375
21.3 Sound chip example program 378
21.4 The speech chip (model B and B+ only) 379
22 User/printer and system VIAs **380**
22.1 6522 Versatile interface adapters in general 380
22.2 The User/Printer VIA 381
22.3 The System VIA 384
22.4 6522 VIAs Functional Description 387
23 The One Megahertz bus & cartridge interfaces **402**
23.1 Introduction to the 1MHz bus 402
23.2 "FRED" and Memory Mapped Hardware 403
23.3 "JIM" and 64K Paged Memory 404
23.4 Bus signal definitions 405
23.6 Hardware requirements for 1MHz bus peripherals 410
23.7 Master, Compact & Electron Cartridge Interface 413
24 Miscellaneous topics **419**
24.1 BREAK 419
24.2 Printer OS calls 422
24.3 *CODE 425
24.4 Miscellaneous OSBYTE calls 427
Glossary **430**
Bibliography **433**

Introduction

The *New Advanced User Guide* is the totally revised and updated version of the original best-selling *BBC Advanced User Guide* by the same authors. In the four years since that book was first published, a lot has happened in the world of Acorn Computers, including the introduction of many new BBC Micro-compatible machines. Based around the original hardware and software of the BBC model B, the new computers have many significant enhancements and changes. This new guide has therefore been developed to cover all the computers in the series from the original model B through the B Plus, Electron and Master 128 to the Master Compact.

The authors have condensed vast amounts of information into these pages covering all aspects of the Acorn-BBC computer range, including many previously unpublished details about the new systems' hardware and software. All the computers are considered in detail throughout the book with salient differences being carefully noted and explained. Readers of the original Advanced User Guide will notice that the New Guide has been re-organised on a functional basis, making reference to particular topics much easier.

For owners of Master series computers this advanced guide covers many areas omitted from Acorn's Master Reference Manuals part 1 and part 2, especially interface details, programming techniques, and examples.

For owners of earlier machines there is a lot of relevant new information about the Tube™ and many more detailed examples covering topics like paged ROMs.

This guide contains a considerable amount of information about assembly language programming, including all the new 65C02 op-codes, the operating system, and the system hardware and interfaces. The intention has not been to provide the inexperienced user with a tutorial to guide him or her through the complexities of these advanced concepts. However, it is hoped that the information has been presented in a way that enables users new to assembly language programming and unfamiliar with hardware topics to develop their understanding of the machine and to expand the scope of their programming. Contained within this book is an extensive description of the software environment and the hardware facilities available to the assembly language programmer. The authors have presumed that the readers of this Advanced Guide are reasonably familiar with the basic use of their BBC Microcomputer. While every attempt has been made not to bury the facts under a mountain of computer jargon, the use of some technical

terms is an inevitable consequence of attempting to condense a large number of facts into a concise and easily accessible form. The extensive glossary of terms should help to unshroud some of the mystery.

While this book gives the programmer full access to all the BBC microcomputer's extensive software and hardware facilities using techniques which the designers of the machine intended programmers to use, it also opens the door to a multitude of *illegal* programming techniques. For the enthusiast, direct access to operating system variables or chip registers may enable him to perform the bizarre or even the merely curious. For the serious programmer, on the other hand, attention to compatibility and machine standards will enable him to write software which will run on BBC Microcomputers of all configurations. The responsibility rests with YOU, the user. The value of your machine depends on continued software support of the highest quality; unlike many machines the BBC microcomputer has been designed to be used in a variety of different configurations and the operating system software provides extensive information about the current hardware and software status. The operating system makes most of the allowances required for the different configurations automatically, but only when the legal techniques are adhered to, so please use them.

1 Introduction for those new to machine code

There comes a time in every programmer's life (well, most programmers', anyway) when the constraints of a high level language (e.g. BASIC) prevents him or her from implementing a particular program idea or from utilising some machine facility. At this stage the programmer must often seek recourse to the microprocessor's native language, its machine code.

At the heart of any microcomputer is the microprocessor. This microprocessor is the brain of the computer and provides the computer with all its computing power. The Acorn BBC range of computers use members of the 6502 family of microprocessors and the brief description of machine code given here applies specifically to the 6502. The microprocessor performs instructions which are contained in memory. Each instruction which the microprocessor understands can be contained within a single byte of memory. Depending on the nature of this instruction the microprocessor may fetch a number of bytes of data from the memory locations following the instruction byte. Having executed this instruction the microprocessor moves on to the byte after the last data byte to get its next instruction. In this way the microprocessor works its way sequentially through a program. These single byte instructions are called *operation codes* (or just opcodes), although they are frequently referred to as machine instructions (or just instructions). The data used by the instructions are called the *operands*. A program using the native machine instructions is called a *machine code* program. An *assembler* is a software package (a language or a program) which enables a programmer to create a machine code program.

Machine code is substantially different from a higher level language such as BASIC. The machine code programmer is limited to three registers for temporary storage of data while in BASIC he has unlimited use of variables. For more permanent storage in machine code programs, the register values can be copied to bytes of memory. Only very limited arithmetic is available; there are no multiply or divide instructions. There are no automatic loop structures such as FOR...NEXT or REPEAT...UNTIL and any loops must be explicitly set up by the programmer using conditional branches (these approximate to IF .. THEN GOTO .. in BASIC). The range of instructions available are sufficient to enable extremely complex programs to be written, but a lot more effort is required to implement the program. One of the most grave disadvantages of machine code is that very little error checking is made available to the programmer. A well designed assembler will help

the programmer, but once the machine code program is running the only error checking is that which is provided within the program itself.

At first glance it may appear that there is little to be gained from writing a machine code program. The principal advantage is that of speed. Whilst assigning a value to a variable in BASIC will take about 1 millisecond, in machine code assigning a similar value will only take 10 microseconds. This is why fast moving arcade games have to be written in machine code. Some of the facilities available on the BBC Microcomputer can only be used when programming in machine code. For example, a user printer driver can only be implemented in machine code.

Of no less importance than the design of the hardware or the choice of microprocessor in the machine is the operating system. This is a large, and highly complex machine code program which governs the machine. The operating system consists of a large number of routines which perform operations such as scanning the keyboard, updating the screen, performing analogue-to-digital conversions (for the joy-sticks), and controlling the sound generator. All these functions are performed by the operating system and are made available to other machine code programs. A machine code program with which all users will be familiar is BASIC. This program, which provides the user with an easier way of using the microprocessor's computing power, constantly uses the operating system routines to get input from the keyboard and to reflect that input on the screen. BASIC recognises words of text and when it wants to use a hardware facility it calls a machine code routine within the operating system. The great advantage of this independence of the language program from the direct use of hardware is that the same facilities can be offered to different languages without re-writing the routine.

Writing a machine code program requires the programmer to place the appropriate values into successive memory locations corresponding to the opcodes and operands. This would be a very tedious business if it had to be done by looking up the opcode values in tables and poking in the values by hand. It is much faster, easier and more efficient to get the computer to do most of the work. A program which analyses text input representing opcode symbols, converts these to opcode values and inserts these values into memory is called an *assembler*. The text input consists of opcode mnemonics (three-letter words which specify the opcode type) followed by numbers, variable names or expressions which give the values of the operand to be used by the opcode. Like BASIC the assembler requires the program to be written in a defined way according to a syntax. The language that an assembler understands is called *assembler* or *assembly* language. In the BBC Microcomputer an

assembler is available as part of the BASIC language and a description of how this assembler can be used is contained in the chapter on the BASIC assembler.

Many of the following sections include descriptions of the various operating system routines and facilities which are available to the machine code programmer.

2 The BASIC Assembler

One of the many attractive features of BBC BASIC is the incorporation of a mnemonic assembler within the language itself. This provides a powerful environment for the assembler and allows machine code to be easily incorporated within BASIC programs. Hybrid BASIC/machine code programs may often lead to the use of the best features of each language, the speed of machine code when it is required, coupled with the increased power of BASIC when speed is not of paramount importance.

For users of older BBC model Bs the assembler facilities available are dependent upon the version of BASIC that is resident in the machine. To ascertain which version of BASIC is present type "REPORT" following a BREAK. If the copyright message is dated 1981 then this is *old BASIC* which will henceforth be referred to as Level 1 BASIC If the message is dated 1982 or later then this is *new BASIC* which will be referred to as Level 2 BASIC. Level 4 BASIC as provided in the Master 128 and Level 40 BASIC as provided in the Master Compact are virtually identical to Level 2 BASIC from an assembly language programmer's point of view. One minor change in level 4 and 40 is that register and EQU references can be entered in lower case.

Users of later BBC model B's, Electrons, 6502 second processors, and Master series computers may ignore any comments specific to Level 1 BASIC.

Below is an example of a simple machine code program written using the BASIC assembler.

```
10 OSASCI=&FFE3
20 DIM MC% 100
30 DIM data &20
40 FOR opt%=0 TO 3 STEP 3
50   P%=MC%
60   [
70   OPT opt%
80   .entry  LDX #0          \ set index count (in X reg.) to 0
90   .loop   LDA data,X      \ load next VDU parameter
100          JSR OSASCI     \ perform VDU command
110          INX            \ increment index count
120          CPX #&20       \ has count reached 32 (&20) ?
130          BNE loop       \ if not then go round again
140          RTS            \ back to BASIC
150  ]
160  NEXT opt%
170 !data=&04190416
180 data!4=&00C800C8
190 data!8=&00000119
200 data!&C=&01190064
210 data!&10=&000000C8
```

```
220 data!&14=&00000119
230 data!&18=&0119FF9C
240 data!&1C=&0000FF38
250 PRINT'"Press key to run program":A=GET
260 CALL entry
```

This program performs some simple graphics using the BASIC VDU method to select the screen MODE and perform PLOTting. All the VDU codes are contained within the block of memory labelled "data". Using the ! operator does not make it immediately obvious what is going on. Four bytes are inserted into memory with each ! operator. The least significant byte being inserted at the address specified. Each subsequent byte is inserted into the next byte of memory. i.e.

```
!data=&04190416
data!4=&00C800C8
```

will have a result equivalent to,

```
VDU &16,&04,&19,&04,&C8,&00,&C8,&00
```

or, to separate it into its two components,

```
VDU &16,&04
VDU &19,&04,&00C8;&00C8;
```

or

```
VDU 22,4              select MODE 4
VDU 25,4,200;200;     PLOT,4,200,200 - move absolute X,Y
```

Any program which can be written in BASIC may also be implemented in machine code although it is not always sensible to do so. There now follows a detailed description of using the BASIC mnemonic assembler.

2.1 The assembler delimiters '[' and ']'.

All the assembler statements should be enclosed within a pair of square brackets. When the BASIC program is RUN, the assembler statements contained between the square brackets are assembled into machine code. This code is inserted directly into memory at the address specified by P%, and P% is incremented by the number of bytes in each instruction or directive. The assembly language program is written between the assembler delimeters. This program will consist of a number of assembler statements separated by new lines or colons (as in BASIC).

Each assembler statement should consist of an optional label followed by an instruction (this will be a three letter assembler mnemonic or an assembler directive) and then an operand (or address). If a label is

included it should be separated from the instruction by at least one space. The operand need not be separated from the instruction. Any character following the operand and separated by at least one space from it will be totally ignored by the assembler which will move onto the next colon or line for the next statement. A comment may be placed after the operand field and should be preceded by a backslash (\). Any text following a backslash in an assembly statement will be ignored by the assembler up to the next colon or end-of-line.

N.B. In level 1 BASIC colons cannot be included in expressions. Missing out a colon in a multi-statement line will result in the statement after the intended colon being ignored by the assembler. This error is often difficult to spot in a program which assembles without error but then fails to function as the programmer had anticipated.

During assembly of the example program the following printout is produced (with `PAGE=&1900`) :

```
>RUN
1BEA
1BEA
1BEA             OPT opt%
1BEA A2 00       .entry  LDX #0     \ set index count (in X reg.) to
1BEC BD 5A 1C    .loop   LDA data,X   \ load next VDU parameter
1BEF 20 E3 FF    JSR OSASCI    \ perform VDU command
1BF2 E8          INX           \ increment index count
1BF3 E0 20       CPX #&20      \ has count reached 32 (&20) ?
1BF5 D0 F5       BNE loop      \ if not then go round again
1BF7 60          RTS           \ back to BASIC
```

location op.code/data label/mnemonic/address comment

2.2 OPT, assembler option selection

OPT is an assembler directive or non-assembling statement which can be included within an assembly program to select a number of different assembler options. The OPT command should be followed by a number to make the option selection. The assembler options are selected on the state of the least significant 2 or 3 bits of the OPT parameter.

bit 0	if set,	assembly listing enabled.
bit 1	if set,	assembler errors enabled.
bit 2	if set,	assembled code placed in memory at O% (Implemented in Level 2 BASIC only)

In the example program, OPT is set up using the FOR..NEXT loop variable, opt%. On the first pass of the assembler OPT 0 is used, listing is suppressed, and assembler errors are not enabled. For the second pass an OPT 3 is used which switches on assembly listing and enables assembler errors. BASIC errors will be flagged as normal. The assembler errors which are suppressed are the "Branch out of range" error and the "No such variable" error. These will normally be generated during the first pass when the assembler is resolving forward passes (see section 2.5). Bit 2 allows a program to be assembled into one region of memory whilst being set up to run at a different address. P%, the program counter (see below) should be set up as usual to provide the source of label values. If bit 2 is set then O% should be set up at the same time as P% to point to the start of memory into which the machine code is to be assembled. This facility is useful for assembling machine code where it is impossible to use the memory in which the program is eventually going to reside (e.g. Assembling programs which are going to be blown into EPROM for paged ROMs). This option is only available in Level 2 BASIC. Each time the assembler is entered the OPT value is initialised to 3. This means that a second chunk of assembler in the same BASIC program must perform its own OPT selection.

2.3 The Location Counter P%

When the assembler creates the machine code program, the code produced is placed in memory starting from the address in P% (one of the resident integer variables) unless remote assembly has been selected using OPT (see section 2.2). The programmer must set P% to a meaningful value before the assembly begins. The usual method for short programs is to DIMension a block of memory and to set P% to this value at the beginning of each pass of the assembler (as in the example). A classic problem is sometimes encountered when a programmer adds more code to a program which has been allocated space by this method. If the code created overflows the space DIMensioned for it, and is over-written by BASIC, it will fail to operate as expected when tested; alternatively the code may over-write the BASIC dynamic storage and a "No such variable" error will be flagged during the second pass of the assembler.

The assembler updates P% as it is assembling and when it reaches the end of a pass the value of P% represents the address of the first 'free' byte of memory after the machine code program.

2.4 Labels

Any BASIC numerically assignable item may be used as a label with the assembler (such as a variable or an array element). A label is defined by preceding the variable name with a full stop. The full stop prefix causes the assembler to set up a BASIC variable containing the current value of P%. Once set up this variable is available for use by any other part of the assembler or BASIC program.

2.5 Forward Referencing and Two Pass Assembly

A large number of labels may be generated during the construction of a machine code program using the BASIC assembler. It is often the case that one part of the program needs to jump forward over another part of the program. Labels provide a convenient way of marking the point in the program to which the processor is to jump. When assembling the machine code, the assembler works sequentially through the program and in the case of a forward reference the assembler will encounter the reference before the label. In the normal course of events an error will be flagged (No such variable). In order to resolve forward references, two passes of the assembler are required. The first pass should be performed with error trapping switched off and during this pass all the labels will be initialised. The second pass will provide all the correct values required for forward referencing. During this second pass error trapping should be enabled to pick up any genuine programming mistakes.

The most convenient way of performing the two passes is to use a FOR...NEXT loop. The programmer should make sure that P% is re-initialised at the beginning of the second pass. It is often convenient to set up the pseudo-operation OPT using the FOR loop variable (errors and listing disabled for the first pass, errors enabled and listing as required for the second).

2.6 The EQUate Facility in Level 2 BASIC

One of the improvements made to Level 2 BASIC was the incorporation of some EQU pseudo-operation commands. These allow the incorporation of data by reserving memory within the body of the assembly language program.

The EQUate operations available are :-

EQUB	*equate byte*	reserves 1 byte of memory
EQUW	*equate word*	reserves 2 bytes of memory

EQUD	*equate double word*	reserves 4 bytes of memory
EQUS	*equate string*	reserves memory as required

These operations initialise the reserved memory to the values specified by the address field. The address field may contain a string in double quotes, a string variable for the EQUS operation, or a number or numeric variable for the other EQU operations. The assembler will use the least significant part of the value if too large a value is specified.

The example program, written in Level 2 BASIC, could have been written with lines 30 and 170 to 240 replaced with :-

```
141 .data EQUD &04190516
142       EQUD &00C800C8
143       EQUD &00000119
144       EQUD &01190064
145       EQUD &000000C8
146       EQUD &00000119
147       EQUD &0119FF9C
148       EQUD &0000FF38
```

In Level 1 BASIC one way to reserve space for data within the body of a machine code program is to leave the assembler using a right-hand square bracket and insert the data using the address contained in P%. P% should then be incremented by the appropriate amount before entering the assembler. e.g. to incorporate a string into a machine code program.

```
 10 DIM MC% 100
 20 OSRDCH=&FFE0
 30 OSASCI=&FFE3
 40 FOR opt%=0 TO 3 STEP3
 50   P%=MC%
 60   [
 70   OPT opt%
 80   .entry LDY #0          \ zero loop index
 90   .loop LDA string,Y     \ load accumulator with Y?string
100   JSR OSASCI             \ write the character
110   INY                    \ increment loop index
120   CMP #&0D               \ is the current character a CR
130   BNE loop               \ if not get the next character
140   JSR OSRDCH             \ get character from keyboard
150   CMP #9                 \ is it the TAB key
160   BNE error              \ if not flag an error
170   RTS                    \ return to BASIC
180   .string
190   ]
200   $P%="Please press the TAB key"
210   P%=P%+LEN($P%)+1
220   [
230   OPT opt%
240   .error BRK             \ cause an error
250   ]
260   NEXT opt%
270 ?P%=&FF
280 P%=P%+1
```

```
290 $P%="Wrong key pressed"
300 ?(P%+LEN($P%))=0
310 CALL entry
```

This program prompts the user to press the TAB key by printing out an message. If the wrong key is pressed an error is flagged.

2.7 Handling errors with BRK

In the example program above the BRK instruction is used to generate an error. The BRK instruction forces an interrupt which is interpreted by the operating system as an error. As part of the error handling in BASIC the programmer can incorporate an error number and an error message into his code to identify the error. The byte in memory following the BRK instruction should contain the error number. The error message string should follow the error number and must be terminated by a zero byte.

The following lines set this up :-

```
240    .error BRK              \ cause an error

270 ?P%=&FF                          Error number 255
280 P%=P%+1
290 $P%="Wrong key pressed"          Error message
300 ?(P%+LEN($P%))=0                 Terminating byte
```

When a BRK is encountered in a machine code program called from BASIC, the error message is printed out together with the line number from which the machine code was called. Typing "REPORT" or printing ERR will reproduce the message and error number as with any BASIC error.

The user can provide his own BRK handling routine which may be useful when using machine code away from the BASIC environment (see section 8.13 for more information about the BRK vector).

2.8 Entering machine code from BASIC - CALL and USR

Machine code routines can be entered from a BASIC program using either the CALL statement or the USR function. On entry to the machine code program using these instructions, the accumulator, the X register, the Y register and the carry flag are set to the least significant bytes (or bit) of the resident integer variables A%, X%, Y% and C%. A number of parameters may be passed to the machine code routine if the CALL

statement is used, the addresses and data types of these parameters being available to the machine code in a parameter block at location &600. The USR function allows the machine code routine to return a value to the BASIC program made up from the register contents. For more details about CALL and USR refer to the 'User Guide' or 'Reference Manual'.

2.9 Conditional Assembly and Macros

When working within the BASIC environment, it is possible to use BASIC functions to implement these higher level assembly language structures.

Conditional assembly is a method of varying the code assembled according to a test. All the facilities of BASIC are available for setting up the test criteria. Typical applications for conditional assembly include the conditional incorporation of debugging routines and selecting different hardware specific sub-routines from a number of alternatives.

A macro is a group of assembler statements which may be inserted into the assembler program whenever the macro is called. A macro may be thought of as being a type of sub-routine which is used to include a portion of assembler used more than once within a program. A number of statements which are likely to be used more than once can be enclosed within assembler delimiters and placed within either a sub-routine (called using GOSUB and terminated by RETURN), or a function definition or a procedure definition. Using a procedure or a function is the best way to implement macros because the programmer is then able to pass parameters to the macro and the procedure/function name serves to identify the macro.

e.g.

```
 10 DIM MC% 100
 20 FOR opt%=0 TO 3 STEP 3
 30   P%=MC%
 40   [
 50   OPT opt%
 60   .add LDA &80
 70   ADC &81
 80   STA &81
 90   OPT FNdebug(TRUE)
100   ]
110   NEXT
120 ?&80=1
130 ?&81=2
140 CALL add
150 PRINT'"Result of addition : ";?&81
160 PRINT'"A=&";~?&70,"X=&";~?&71,"Y=&";~?&72
170 END
```

```
180 DEF FNdebug(switch)
190 IF switch [OPT opt%:STA &70:STX &71:STY &72:]
200 [OPT opt%:RTS:]
210 =opt%
```

This highly contrived program adds two bytes together. It uses a macro within which conditional assembly occurs. Hanging a function on the end of an OPT command enables the programmer to call the macro in a tidy manner. If FNdebug is called with the value TRUE then some code which saves the registers in zero page is inserted into the program, otherwise an RTS instruction is inserted. The function returns with the value to which OPT was originally set. This example indicates how the close relationship between the mnemonic assembler and BASIC results in a very powerful assembler. The programmer should remember that BASIC is always available as an aid when using the BASIC assembler.

2.10 User Zero Page

32 bytes of zero page locations are reserved by BASIC for the user's machine code programs. These locations are from &70 to &8F (inclusive). These are the only zero page locations that a user program (resident in RAM) should use if the program is to be made commercially available or run on a variety of other BBC Microcomputers. The locations from &0 to &6F which are part of BASIC's zero page workspace are available to the machine code program if BASIC is not required while the code is running. Depending on the nature of the machine code program other zero page locations may be available. See chapter 12, memory usage, for more details.

3 Machine Code Arithmetic

3.1 2's Complement

The 6502 microprocessor normally performs all arithmetic using the 2's complement method of representing numbers. In 2's complement representation the most significant bit of the value is a sign bit. If the most significant is clear then the number is positive. The remaining bits represent the binary value of the positive number. Negative values are represented by the complement of the positive value plus 1. The complement of any binary value is made by *flipping* each bit (i.e. changing each 1 to a 0 and each 0 to a 1). When negative values are represented by the complement of the positive value this is called 1's complement. The disadvantage with 1's complement is that there are two ways of representing 0, a positive 0 (all bits clear) and a negative 0 (all bits set). By adding one to the complemented value (2's complement) there is only one way of representing 0 (all bits clear).

e.g. Using 8 bits to store a value

binary 5	0	0	0	0	0	1	0	1
complement of 5	1	1	1	1	1	0	1	0
add 1								1
2's complement (-5)	1	1	1	1	1	0	1	1

Such that the addition of plus 5 and minus 5 yields a result of zero

(-5)	1	1	1	1	1	0	1	1	+
(+5)	0	0	0	0	0	1	0	1	
(0)	0	0	0	0	0	0	0	0	

(ignore the carry from the last bit)

Numbers from 1 0 0 0 0 0 0 0 (-128) to 0 1 1 1 1 1 1 1 (+127) can be represented using 8 bit 2's complement values.

Using 2's complement arithmetic the same addition and subtraction operations work identically on negative and positive numbers. Negative numbers can always be recognised by the state of the most significant bit; this is set for negative numbers.

The 6502 microprocessor can only perform its arithmetic operations using 8 bit values. This limitation can lead to errors when a carry is generated on the most significant bit so that the result cannot be stored in 8 bits. The sign bit may also be wrongly changed when a carry occurs into it. Two flags in the status register are set when certain conditions occur. These flags are the carry flag and the overflow flag.

The carry flag is set when a carry is generated during an addition operation if a carry is generated from bit 7 (i.e. the carry flag is a ninth bit of the result). The carry flag is cleared if a borrow occurred into bit 7 during a subtraction. The addition and subtraction instructions on the 6502 include the carry bit in the operation. Using the carry bit makes it possible to perform multi-byte arithmetic. The examples for ADC and SBC in the mnemonics section illustrate how the carry flag may be used.

The overflow flag is set when the sign of the result is incorrect following an arithmetic operation. During additions overflow will occur in two situations :-

(a) When a carry occurs from bit 6 into bit 7 without the generation of an external carry.

(b) When an external carry is generated without a carry occurring from bit 6 into bit 7.

During subtractions the carry flag is used as a borrow source. The overflow flag will be set in the analogous situations where borrows occur rather than carries. When the overflow flag is set it indicates that the 2's complement 8 bit result of an arithmetic operation is incorrect.

It is often more convenient to think of bytes as always containing positive values. The eight bits of the byte can represent a maximum binary value of 255 (&FF). This is not a problem because the microprocessor performs exactly the same arithmetic operations regardless of the sign of the values involved. When the result of any arithmetic operation has bit 7 set then a negative flag is set in the status register. The programmer can test this flag if the program must react to negative values. The overflow and carry flags will also be set as described above.

3.2 Binary Coded Decimal

A binary coded decimal arithmetic mode may be selected by setting the decimal flag in the status register. The binary coded decimal form of representing numbers uses each byte to store a two digit decimal value. Each digit is stored as a binary value in 4 bits (1 nibble). Normally 4 bits can be used to represent numbers in the range 0 to 15. In BCD arithmetic 6 of the values that could be represented in 4 bits are not used. Adding 1 to 9 in BCD will cause the low-nibble to be set to 0 and the high nibble to be set to 1. The carry flag is used to store the carry from the high-nibble.

This is an example of a program which uses BCD arithmetic.

```
 10 DIM MC% 100
 20 OSWRCH=&FFEE
 30 OSRDCH=&FFE0
 40 OSNEWL=&FFE7
 50 FOR opt%=0 TO 3 STEP3
 60   P%=MC%
 70   [
 80   OPT opt%
 90   .start SED    \ set flag for BCD arithmetic
100   CLC           \ clear carry flag
110   LDA &80       \ A=?&80
120   ADC #1        \ A=A+1+C
130   STA &80       \ replace value
140   LDA &81       \ A=?&81
150   ADC #0        \ A=A+0+C
160   STA &81       \ replace value
170   CLD           \ clear flag, no more BCD
180   CLC           \ clear carry flag
190   LDX #2        \ set loop index
200   .loop DEX     \ decrement index
210   LDA #&F0      \ mask for high-nibble
220   AND &80,X     \ A=A AND X?&80
230   LSR A:LSR A:LSR A:LSR A
240                 \ move high-nibble to low nibble
250   ADC #&30      \ add value to ASC"0"
260   JSR OSWRCH    \ print value
270   LDA #&F       \ mask for low-nibble
280   AND &80,X     \ A=A AND X?&80
290   ADC #&30      \ add value to ASC"0"
300   JSR OSWRCH    \ print number
310   CPX #0        \ has index reached 0
320   BNE loop      \ if not, go round again
330   LDA #&D       \ A=carriage return value
340   JSR OSWRCH    \ perform carriage return (no LF)
350   JSR OSRDCH    \ A=GET
360   CMP #&0D      \ was it RETURN
370   BNE start     \ if not, back to the start
380   JSR OSNEWL    \ carriage return and line feed
390   RTS           \ back to BASIC
400   ]
410   NEXT
420 !&80=0
430 PRINT''"Binary Coded Decimal"''
440 PRINT"press key to add 1"
```

```
450 PRINT"press RETURN to exit"''
460 CALL start
```

This program could be altered to subtract 1 each time a key is pressed by changing line 100 to SEC and changing the ADC instructions in lines 120 and 150 to SBC instructions.

The decimal flag must always be cleared before using operating system routines.

There is no standard representation of negative numbers using BCD. In order to implement more complex arithmetic including floating point applications, the programmer must define his own conventions and number formats.

3.3 BCD and the 65C12

Arithmetic operations performed in decimal mode on the 65C12 microprocessor used in the BBC Master series computers use up more processor time than the same operations performed in non-decimal mode. This is because the 6502 left the N, V and Z flags in an indeterminant state following instructions in BCD mode, whislt the 65C12 chip updates the flags correctly. The ADC and SBC instructions therefore take one extra clock cycle when the decimal flag is set.

4 Addressing Modes

When an assembly language instruction needs an address or some data, this must be provided in the operand field of the assembler statement. Although there are a limited number of different machine code instructions which can be used with the 6502, the power of the instruction set is enhanced by a number of different addressing modes by which the data or addresses used by each instruction may be provided. The addressing mode used by the assembler depends on the syntax of the assembly language statement. The following text describes how the different addressing modes work and the assembler syntax which is necessary.

N.B. Not all addressing modes are available for all instructions. Details of which addressing modes can be used with which instructions are contained in the Assembler Mnemonics section 5.2.

4.1 Implicit addressing

Many instructions do not require any addressing mode to be specified in the operand field. In such cases the addressing is implicit in the instruction itself. For example an RTS instruction will always cause the processor to jump to the location addressed by the top two bytes of the stack.

4.2 Accumulator addressing

Some instructions may operate on either a memory location or the accumulator. The accumulator is specified by putting a capital A in the operand field.

e.g.

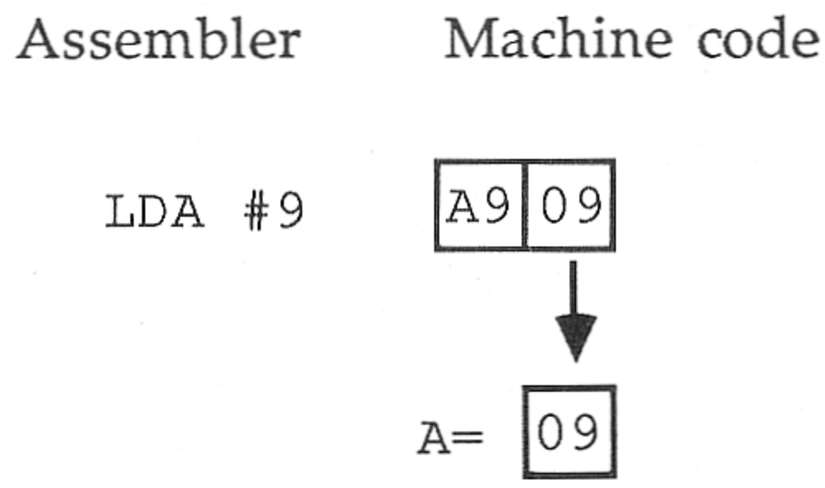
```
        ASL A           \ shift accumulator contents one bit left
        ROR A           \ rotate accumulator contents one bit right
```

(Note that the variable A cannot therefore be used as an operand.)

4.3 Immediate addressing - using a data constant

If, at the time of programming, the data required for a machine code instruction is known then immediate addressing may be used. Immediate addressing is indicated to the assembler by preceding the

operand with a "#" character. The assembler uses the least significant byte of the value given to define the operand. The machine code instruction actually uses the byte of data immediately following the instruction in program memory.

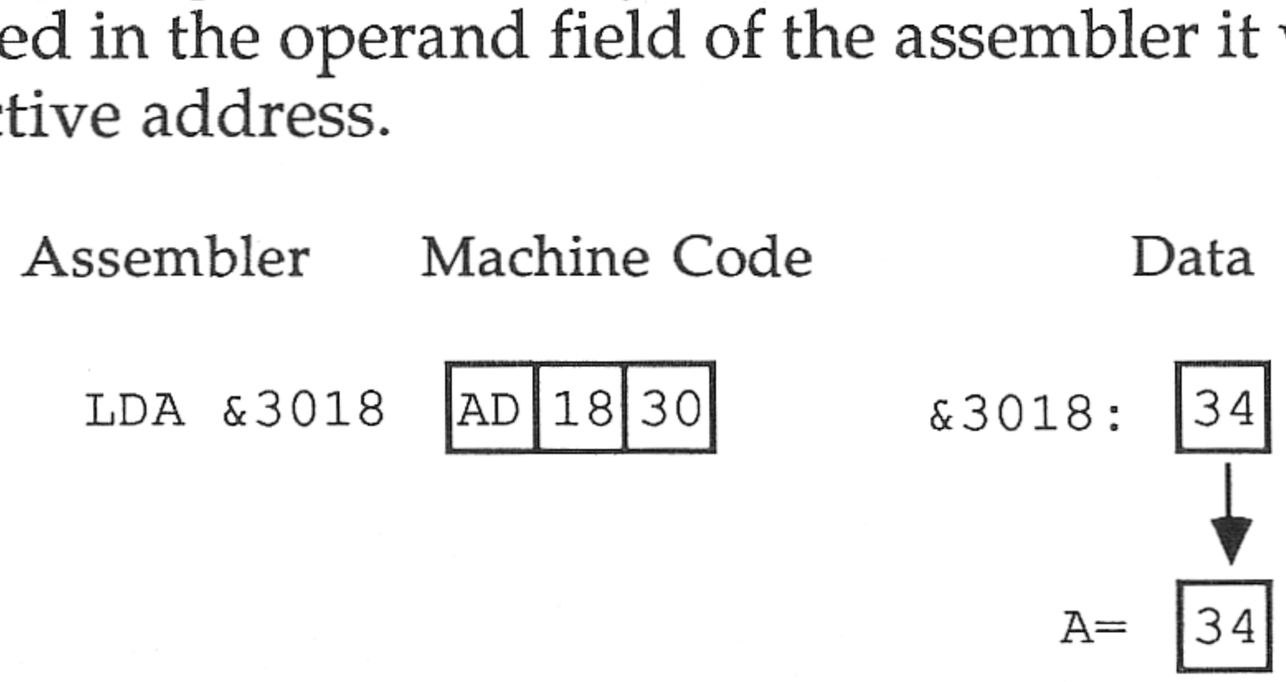


e.g.

```
LDA #&FF      \ load the accumulator with value &FF
LDX #count    \ load X with value of the constant 'count'
```

4.4 Absolute addressing - using a fixed address

When the address required for an instruction is known at the time of assembly then absolute addressing may be used. Absolute addressing is the default addressing mode used by the assembler. If a number or variable is placed in the operand field of the assembler it will be treated as a 16 bit effective address.

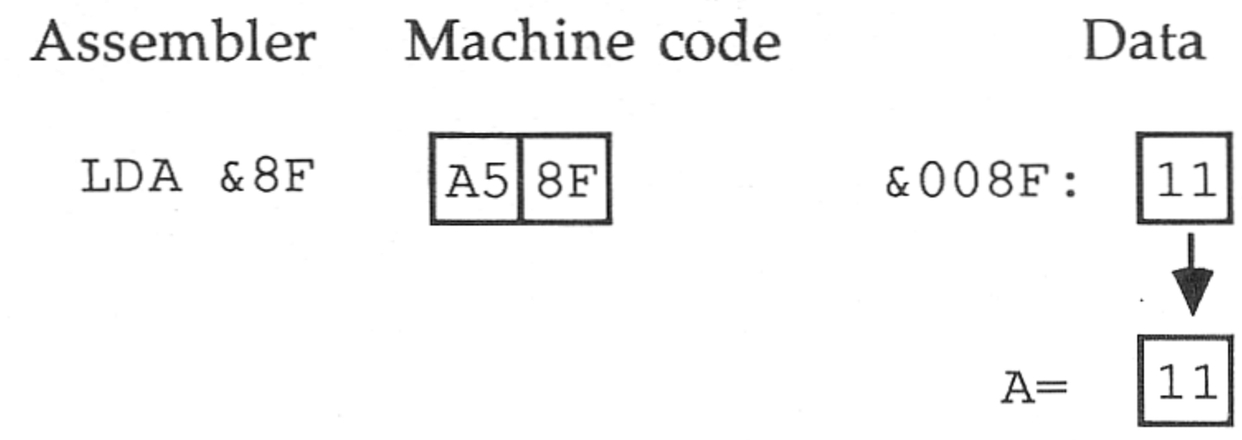


e.g.

```
CMP &1900     \ compare A with contents of location &1900
JMP label     \ goto address specified by 'label'
```

4.5 Zero page addressing - using a fixed zero page address

This mode is the same as absolute addressing except that an 8 bit address is specified. This 8 bit addressing limits use to the first &100 bytes of memory (zero page). The assembler will automatically select zero page addressing when the operand value is less than 256 (&100).

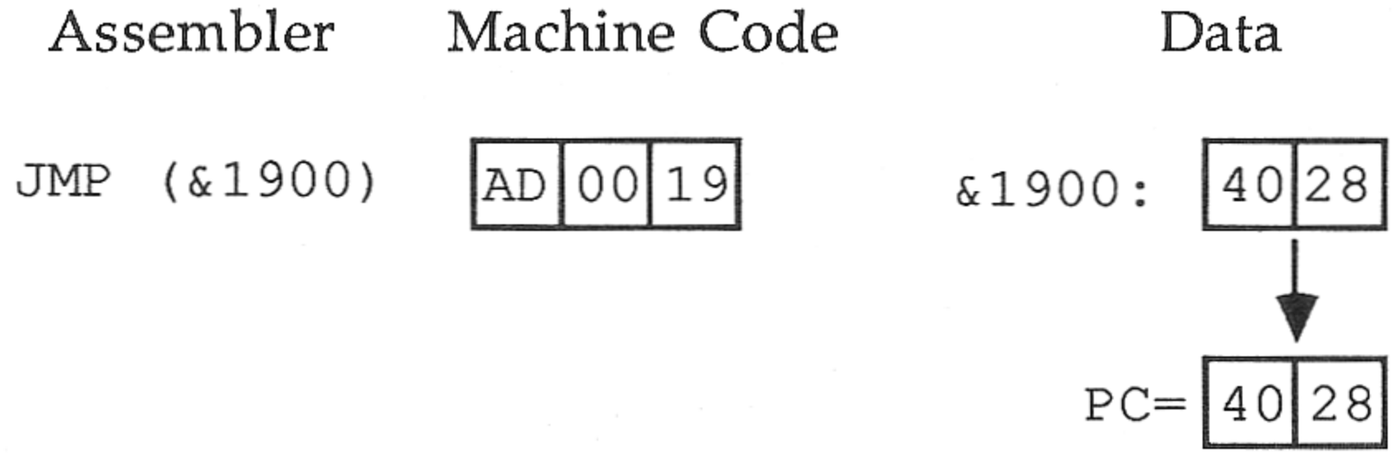


e.g.

```
CPY &80    \ compare Y with contents of location &80
ASL &81    \ shift left contents of location &81 one bit
```

4.6 Indirect addressing - using an address stored in memory

Using this addressing mode an instruction can use an address which is computed when the program runs. The JMP instruction may use this addressing mode. The address used for the jump is taken from the two bytes in memory starting at the address specified in the operand field (low byte first, high byte second). Indirect addressing is indicated to the assembler by enclosing the address within brackets.



e.g.

```
LDA #&40     \ load accumulator with &40
STA &1900    \ store low byte of indirection
LDA #&28     \ load accumulator with &28
STA &1901    \ store high byte of indirection
JMP (&1900)  \ goto address in &1900 and &1901
               \ i.e. goto &2840
```

N.B. A JMP &2840 instruction would have been more sensible in this case.

There is a *bug* in the 6502a used in the BBC model B and Electron. When the indirect address crosses a page boundary the 6502 does not add the carry to calculate the address of the high byte.

i.e. `JMP (&19FF)` will use the contents of `&19FF` and `&1900` for the `JMP` address.

This bug has been fixed in versions of the 6502 used in the Master series of computers.

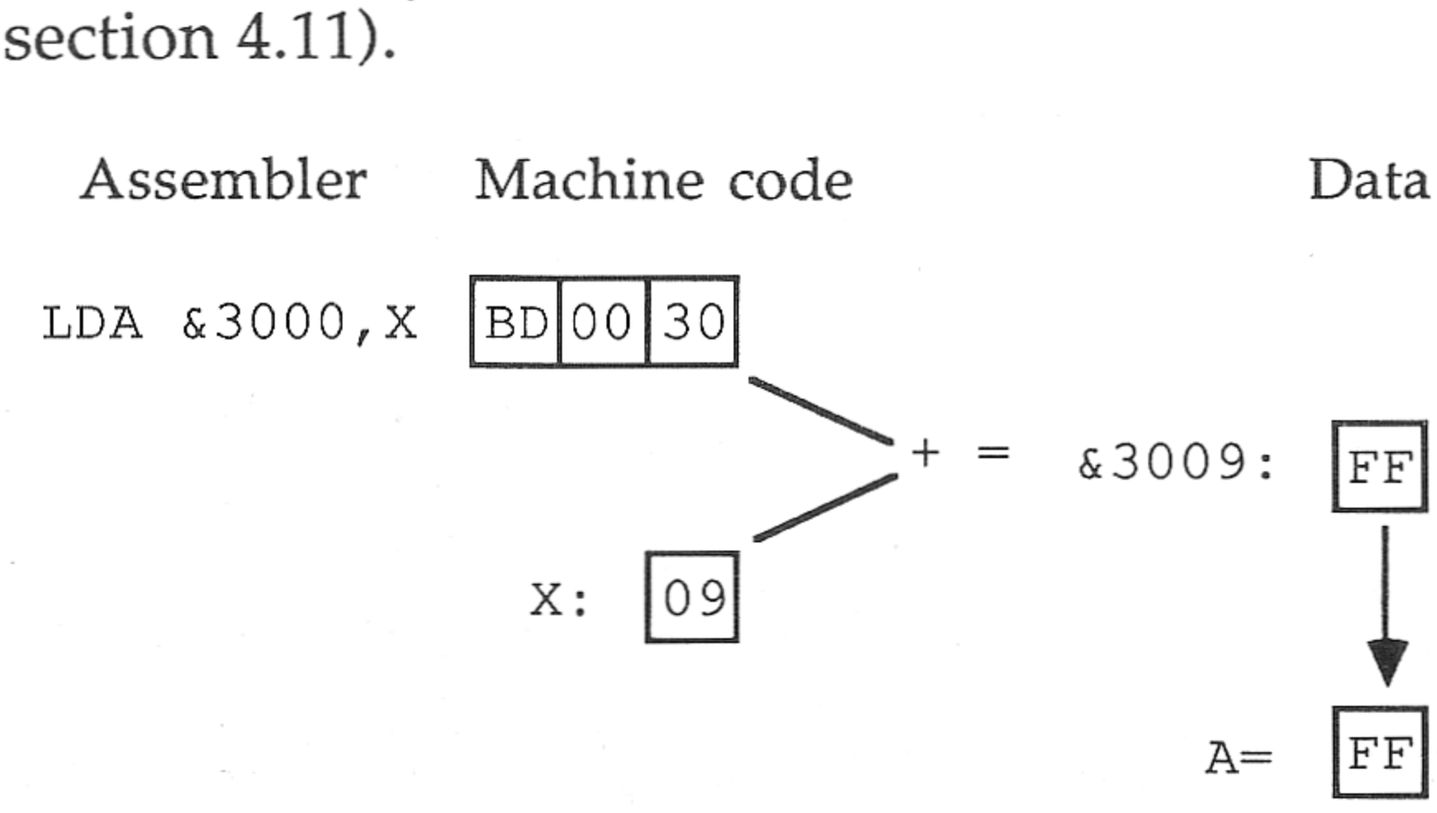
Indexed Addressing

The following 5 addressing modes use the X or Y registers as an offset which is used to modify another address specified in the operand field. These addressing modes give the program access to a table of memory locations specified in terms of a base address to which is added the 8 bit offset value.

4.7 Absolute,X or Y addressing - using an absolute address+X

These are the simplest indexed addressing modes. An absolute 16 bit address is specified in the operand field. This should be followed by a comma and either X or Y. The address used by the instruction will be the 16 bit address + the contents of the register specified.

The X and Y register contents are always taken as positive values in the range 0 to 255 and so only forward offsets are available (c.f. Relative addressing, section 4.11).



e.g.

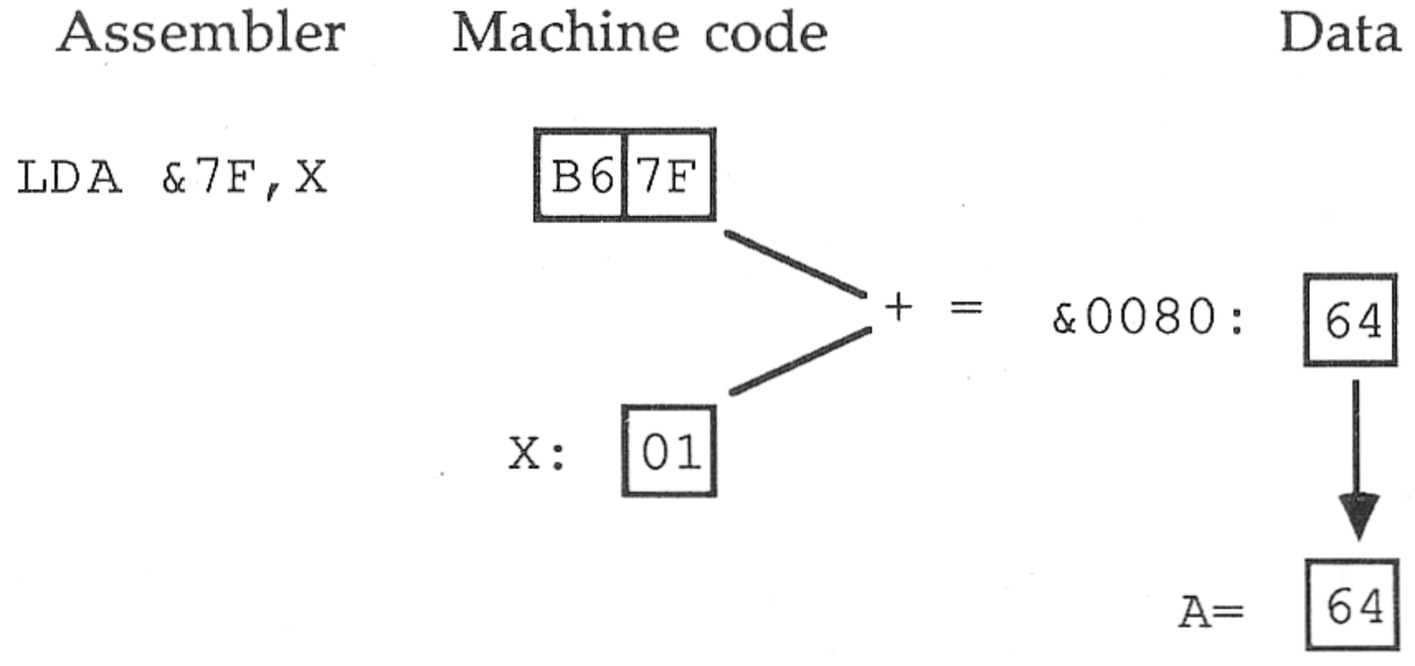
```
LDA &2800,X  \ load accumulator from &2800+X
ADC table,Y  \ A=A+?(table+Y)
```

4.8 Zero page,X addressing - using zero page address+X

This mode is the same as the absolute X addressing mode except that an 8 bit base address is used. The assembler automatically uses this mode, where available, if a zero page address is specified in the operand field.

If a variable is used to describe the address of the zero page location it should be set up before the first pass of the assembler. This is because the assembler will assume 16 bit addressing on the first pass if the variable is unrecognised and allocate two bytes for the address. On the second pass, the zero-page opcode and one byte of address will be assembled, causing all further label values to be wrong.

N.B. For the LDX instruction a zero page,Y addressing mode is provided.



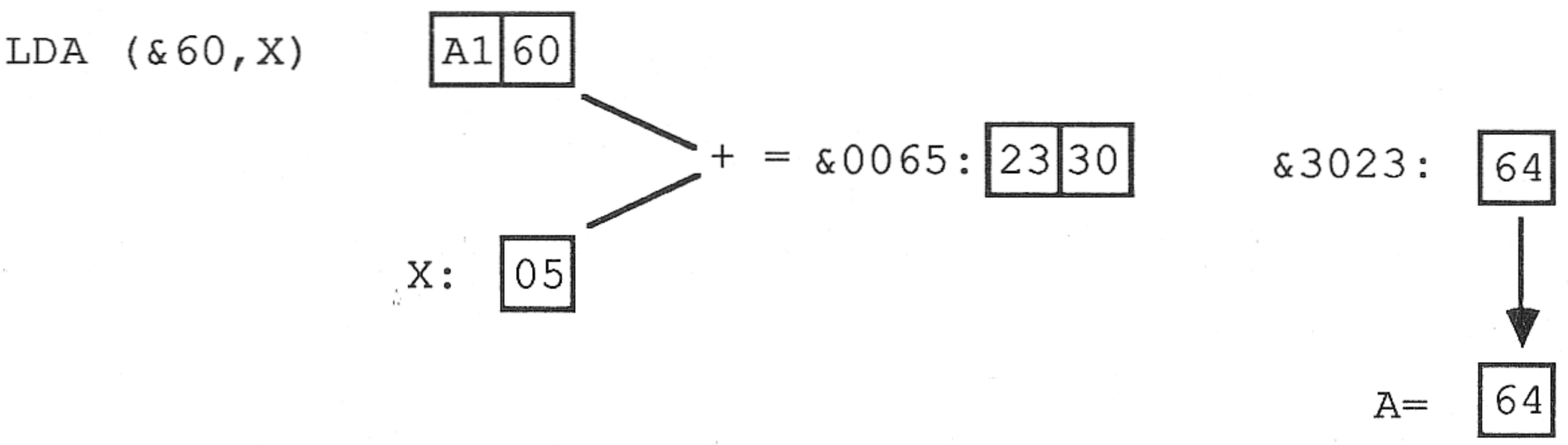
e.g.

```
LDX &72,Y    \ load X with contents of (&72+Y)
LSR &80,X    \ one bit right shift contents of (&80+X)
```

4.9 Pre-indexed indirect addressing using a table of indirect addresses in zero page

This addressing mode is designed for use with a table of addresses in zero page locations. The operation is performed on a memory location, the address of which is contained within the zero page locations specified by an 8 bit base address plus the contents of the X register.

N.B. The Y register cannot be used for this addressing mode.



e.g.

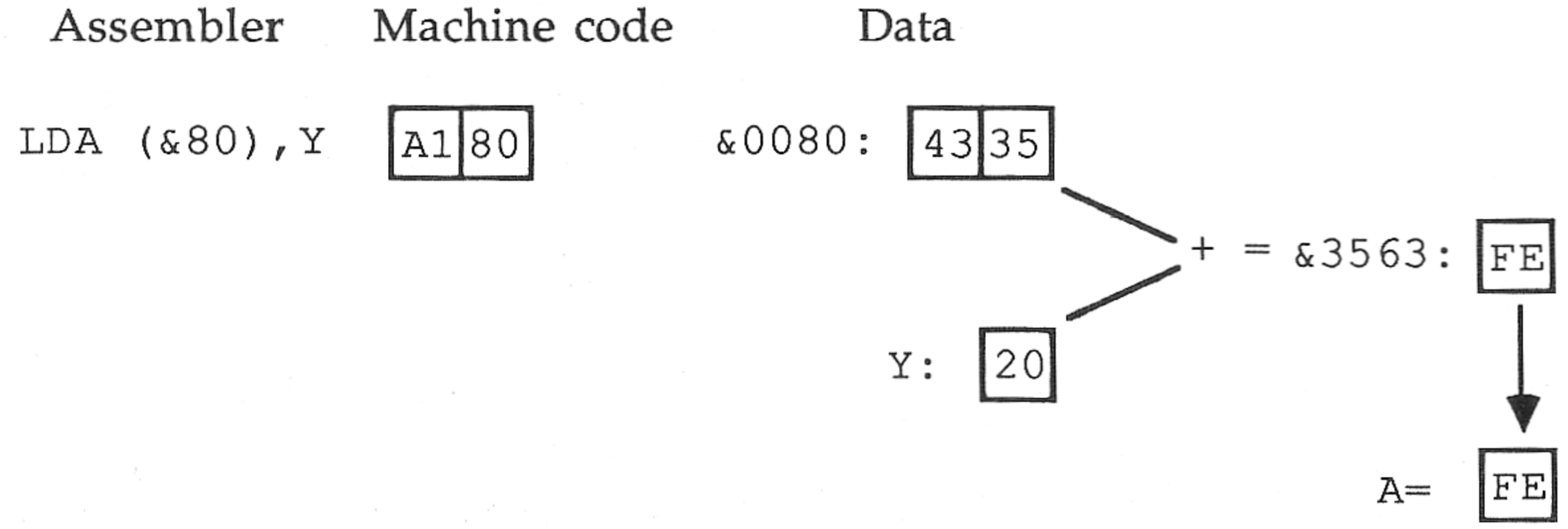
```
?&80=&00
?&81=&40
?&82=&00
?&83=&41

LDX #0          \ set X to 0
LDA (&80,X)  \ A=?&4000, address in (&80+X),(&81+X)
INX             \ X=X+1, i.e. 1
INX             \ X=X+1
LDA (&80,X)  \ A=?&4100, address in (&82),(&83)
```

4.10 Post-indexed indirect addressing using an indirect address in zero page plus offset in Y

This indexed indirect addressing mode uses a single address held in zero page. The contents of the Y register are then added to the address held in zero page to give the effective address used.

N.B. The X register cannot be used for this addressing mode.



e.g.

Set 256 bytes of memory to 0 starting at the address contained in locations &80 (low byte) and &81 (high byte).

```
?&80=&40
?&81=&72

        LDY #0          \ set loop index to 0
        TYA             \ A=0
.loop   STA (&80),Y  \ ?(&7240+Y)=0, base addr. in &80 and &81
        INY             \ Y=Y+1
        CPY #0          \ Y-0 comparison (not needed after INY)
        BNE loop        \ if Y<>0 goto loop
```

4.11 Relative addressing

The 6502 instruction set contains 8 branch instructions which cause jumps if certain conditions are met. In the example above a BNE instruction is used to cause the loop to be executed again if the loop index (Y register) does not equal 0. These branch instructions can only be used with relative addressing. If the condition of the branch is satisfied, the byte following the branch instruction is added to the program counter as an 8 bit two's complement number. This method of relative addressing allows a branch forward 127 bytes or back 128 bytes from the program counter value after the branch instruction has been executed. The calculation of the relative branch value is normally quite transparent to the programmer using the BASIC assembler. When writing in assembly language the programmer follows the branch instruction with a label or absolute address and the assembler performs the necessary calculations. The use of relative addressing will only become apparent when a label or absolute address is specified outside the relative addressing range. When this occurs the assembler will flag an "Out of range" error to the user. OPT 0 is used to suppress this error from forward references on the first assembler pass.

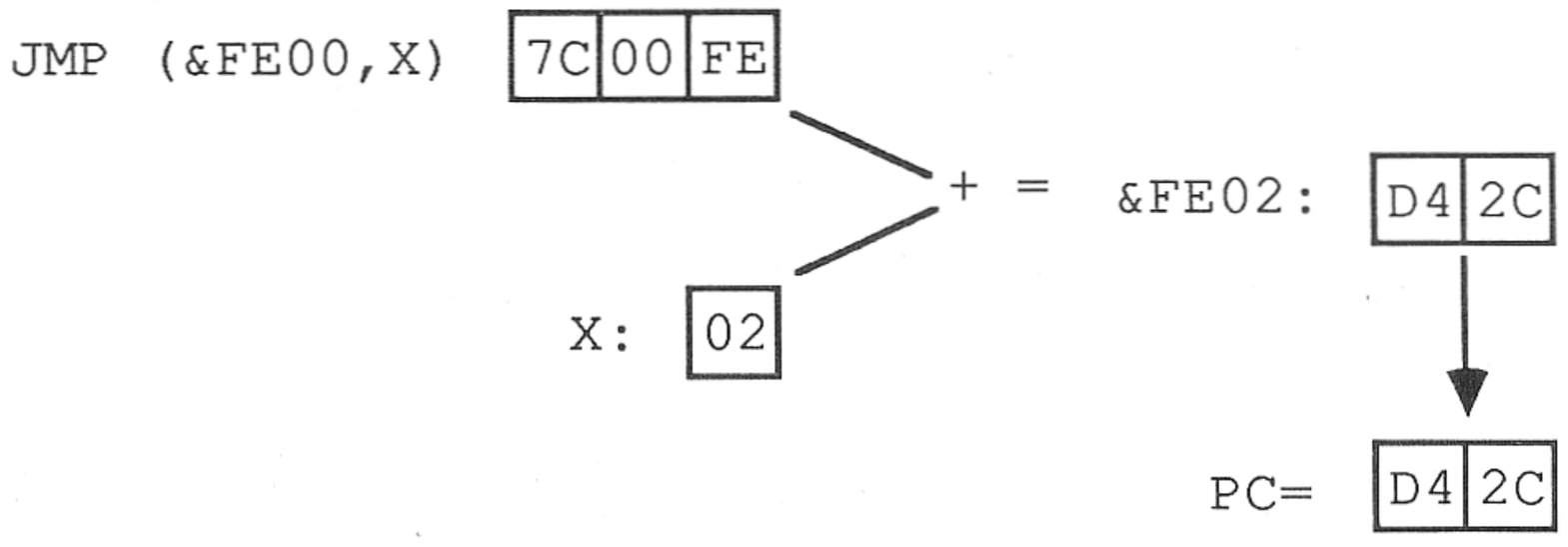
If calculating relative branches by hand remember that the offset value is calculated from the beginning of the next instruction. Thus an offset address of -2 (&FE) will cause a branch back to the branch instruction itself.

4.12 Addressing modes and the 65C12

The microprocessor used in the BBC Master series computers has an improved repertoire of addressing modes. Two additional addressing modes have been added, one allows indirect jumps to be indexed and the other provides indirect zero-page addressing without using an index register. The range of addressing modes available for some instructions has also been expanded.

4.12.1 Pre-indexed absolute indirect addressing

This addressing mode is only available for the JMP instruction. It is identical to the indirect addressing mode except that the the value held in the X register is added to the 16 bit absolute address in the operand field to give the final address containing the value to jump to.



e.g

```
        LDA n           \ A=number indicating a sub-routine
        ASL A           \ A=A*2
        TAX             \ X=A
        LDA #rts        \ A=low byte of address following JMP
        PHA             \ push return address on stack
        LDA #rts/256    \ A=high byte of address following JMP
        PHA             \ push return address on stack
        JMP (&FE00,X)   \ goto 'n'th address in table at &FE
.rts    ...             \ an RTS instruction will return here
```

4.12.2 Zero page indirect addressing

An address stored in zero page memory is used as the working address when this addressing mode is specified. On the 6502 used in earlier Acorn machines the programmer was restricted to the indexed zero page addressing modes for indirect addressing. This mode provides a simple indirect addressing mode which does not tie up one of the index registers.

e.g.

```
        LDA (&86)     \ A=?(!(&86 AND &FFFF))
```

Zero page indirect addressing is available for the following instructions:

ADC	CMP	ORA
AND	EOR	SBC
BIT	LDA	STA

4.12.3 Improved range of addressing modes for some instructions

The range of BIT instruction addressing modes has been extended to include immediate addressing, absolute-X and the new zero page indirect mode.

Accumulator addressing has been extended for use with the DEC and INC instructions. The BASIC assembler recognises the mnemonics DEA and INA as synonyms for the accumulator addressing modes for these instructions.

i.e.

```
DEC A     or   DEA
INC A     or   INA
```

5 The 6502 Instruction Set

This chapter contains a brief description of the 6502 microprocessor registers and a description of the assembly language mnemonics which represent the microprocessor's instruction set.

5.1 The 6502 registers and abbreviations

Accumulator - A

An 8 bit general purpose register used for all the arithmetic and logical operations.

X Index Register - X

An 8 bit register used as the offset in indexed and pre-indexed indirect addressing modes, or as a counter.

Y Index Register - Y

An 8 bit register used as the offset in indexed and post-indexed indirect addressing modes, or as a counter.

Status Register

An 8 bit register containing various status flags and an interrupt mask.

These are :-

Carry flag - C

Bit 0, Set if a carry occurs during an add operation and cleared if a borrow occurs during subtraction. Used as a 9th bit in rotate and shift operations.

Zero flag - Z

Bit 1, Set if the result of an operation is zero, otherwise cleared.

Interrupt disable - I

Bit 2, When set, IRQ interrupts are disabled. Set by the processor during interrupts.

Decimal mode flag - D

Bit 3, When set the add and subtract instructions work in binary coded decimal arithmetic. When clear these operations are performed using binary arithmetic.

Break flag - B

Bit 4, This flag is set by the processor during a BRK interrupt. Otherwise this flag is clear.

Unused flag

Bit 5, Unused by the processor.

Overflow flag - V

Bit 6, If, during an operation, there is a carry from bit 6 to bit 7 and no external carry then the overflow flag is set. This flag is also set if there is no carry from bit 6 to bit 7 but there is an external carry.

Negative flag - N

Bit 7, Set if bit 7 of a result is set, otherwise cleared.

Stack Pointer - SP

An 8 bit register which forms the low order byte of the address of the next free stack location (the high order byte of this address is always &1).

Program Counter - PC (PCL,PCH low-byte,high-byte)

A 16 bit register which always contains the address of the next instruction to be executed.

5.2 The Assembler Mnemonics

The following section contains a detailed description of each of the operation codes (or instructions) in the 6502 instruction set. The assembler recognises three letter mnemonics which it translates into the 8 bit values which the microprocessor actually takes as its instructions.

Each assembler mnemonic is described on a new page. At the head of the page is the three letter mnemonic which the assembler recognises.

Beneath the heading there is a short phrase indicating the function of the instruction and the derivation of the mnemonic.

A shorthand 'BASIC like' description of the operation is given on the top right of the page. The registers and flags are denoted by the abbreviations given on the previous two pages. The initial 'M' represents the data byte obtained using the selected addressing mode.

A brief description of the instruction and its operation is given beneath the headings.

Any changes to the status register are noted in a list of the status register flags.

All the available addressing modes are listed together with the number of bytes of memory which the instruction and its data will occupy when this mode is used. The number of instruction cycles taken for the execution of the instruction in each addressing mode is also given (1 instruction cycle=0.5 µs in a BBC model B, 0.33µs in Master or 6502 2nd processor, 0.25µs in a Master Turbo 2nd processor).

A short example of the use of the instruction within an assembly language routine is given at the bottom of each page.

* New opcodes and addressing modes available on the 65C12 used on the Master series computers are marked by a single asterisk.

** Two additional instructions available on the Rockwell R65C02, used on the Master Turbo and 6502 second processors. These are marked with two asterisks.

Summary of Assembler Mnemonics

ADC	Add with Carry	A,C=A+M+C
AND	Logical AND	A=A AND M
ASL	Arithmetic Shift Left	M=M*2, C=M7
BCC	Branch on Carry Clear	Branch if C=0
BCS	Branch on Carry Set	Branch if C=1
BEQ	Branch on result zero	Branch if Z=1
BIT	Test memory bits	A AND M, N=M7, V=M6
BMI	Branch if negative flag set	Branch if N=1
BNE	Branch on result not zero	Branch if Z=0
BPL	Branch on positive result	Branch if N=0
BRA*	Branch always	
BRK	Forced interrupt	PC and P pushed on stack, PCL=?&FFFE, PCH=?&FFFF
BVC	Branch if overflow clear	Branch if V=0
BVS	Branch if overflow set	Branch if V=1
CLC	Clear carry flag	C=0
CLD	Clear decimal flag	D=0
CLI	Clear interrupt disable flag	I=0
CLR*	Clear memory	M=0
CLV	Clear the overflow flag	V=0
CMP	Compare memory and accumulator	A-M
CPX	Compare memory with X register	X-M
CPY	Compare memory with Y register	Y-M
DEC/A*	Decrement memory by one	M=M-1
DEX	Decrement X register by one	X=X-1
DEY	Decrement the Y register by one	Y=Y-1
EOR	Exclusive OR	A=A EOR M
INC/A*	Increment memory by one	M=M+1
INX	Increment X register by one	X=X+1
INY	Increment the Y register by one	Y=Y+1
JMP	Jump to new location	PC = new address
JSR	Jump Subroutine	Push PC onto stack, PC = new address
LDA	Load accumulator from memory	A=M
LDX	Load X register from memory	X=M
LDY	Load Y register from memory	Y=M
LSR	Logical Shift Right by one bit	M=M/2 (or A)
NOP	No operation	
ORA	OR memory with accumulator	A=A OR M
PHA	Push accumulator onto stack	Push A
PHP	Push Status register onto stack	Push P
PHX*	Push X register onto stack	Push X
PHY*	Push Y register onto stack	Push Y
PLA	Pull accumulator off stack	Pull A
PLP	Pull status register off stack	Pull P
PLX*	Pull X from stack	Pull X
PLY*	Pull Y from stack	Pull Y
ROL	Rotate one bit left	M=M*2, M0=C, C=M7 (A or M)
ROR	Rotate one bit right	M=M/2, M7=C, C=M0 (A or M)
RTI	Return from Interrupt	Status register and PC pulled from stack
RTS	Return from subroutine	Pull PC from stack
SBC	Subtract memory from A with carry	A,C=A-M-(1-C)
SEC	Set carry flag	C=1
SED	Set decimal mode	D=1
SEI	Set interrupt disable flag	I=1
STA	Store accumulator contents in memory	M=A
STX	Store X contents in memory	M=X
STY	Store Y contents in memory	M=Y
STZ*	Clear memory	M=0
TAX	Transfer A to X	X=A
TAY	Transfer A to Y	Y=A
TRB*	Test and reset bits	
TSB*	Test and set bits	
TSX	Transfer S to X	X=S
TXA	Transfer X to A	A=X
TXS	Transfer X to S	S=X
TYA	Transfer Y to A	A=Y

ADC

Add with Carry A,C=A+M+C

This instruction adds the contents of a memory location to the accumulator together with the carry bit. If overflow occurs the carry bit is set, this enables multiple byte addition to be performed.

Processor Status after use

C (carry flag) : set if overflow in bit 7
Z (zero flag) : set if A=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : set if sign bit is incorrect
N (negative flag) : set if bit 7 set

Addressing mode	Bytes used	Cycles	Op. code
immediate	2	2	&69
zero page	2	3	&65
zero page,X	2	4	&75
(indirect zero page)*	2	5 (+1 if in decimal mode)	&72
absolute	3	4	&6D
absolute,X	3	4 (+1 if page crossed)	&7D
absolute,Y	3	4 (+1 if page crossed)	&79
(indirect,X)	2	6	&61
(indirect),Y	2	5 (+1 if page crossed)	&71

Example : Add 1 to a 2 byte value in locations &80 and &81

```
CLC             \ clear carry flag
LDA #1          \ load accumulator with 1
ADC &80         \ A=A+?&80, carry set if overflow occurs
STA &81         \ place result of addition in &80
LDA #0          \ set accumulator to 0 (carry unchanged)
ADC &81         \ A=A+?&81+C, add 1 if carry set
STA &81         \ store result back in &81
```

AND

Logical AND — A=A AND M

A logical AND is performed, bit by bit, between the accumulator contents and the contents of a byte of memory. The truth table for the logical AND is :-

Acc. bit	Mem. bit	Result bit
0	0	0
0	1	0
1	0	0
1	1	1

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if A=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 set

Addressing mode	Bytes used	Cycles	Op. code
immediate	2	2	&29
zero page	2	3	&25
zero page,X	2	4	&35
(zero page indirect)*	2	5	&32
absolute	3	4	&0E
absolute,X	3	4 (+1 if page crossed)	&3D
absolute,Y	3	4 (+1 if page crossed)	&39
(indirect,X)	2	6	&21
(indirect),Y	2	5 (+1 if page crossed)	&31

Example : Clear the bottom 4 bits of location &80

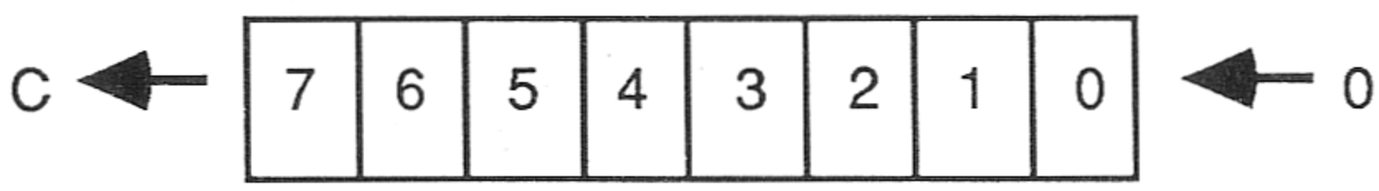
```
LDA &80       \ load value to be ANDed into A
AND #&F0      \ perform AND, (mask=11110000)
STA &80       \ load memory with the modified value
```

ASL

Arithmetic Shift Left

M=M*2, C=M7 (or accumulator)

This operation shifts all the bits of the accumulator or memory contents one bit to the left. Bit 0 is set to 0 and bit 7 is placed in the carry flag. The effect of this operation is to multiply the memory contents by 2 (ignoring 2's complement considerations), setting the carry if the result will not fit in 8 bits.



Processor Status after use

C (carry flag) : set to old contents of bit 7
Z (zero flag) : set if result=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of the result is set

Addressing mode	Bytes used	Cycles	Op. code
accumulator	1	2	&0A
zero page	2	5	&06
zero page,X	2	6	&16
absolute	3	6	&0E
absolute,X	3	7	&1E

Example : Rapid multiplication of memory contents by 4

```
ASL data      \ ?data=?data*2
ASL data      \ ?data=?data*2, gross effect *4.
```

BBR**

Branch on Bit Reset Branch if bit=0

BBR is only available on the Rockwell R65C02 microprocessor normally fitted to the Master Turbo or 6502 2nd processors.

The branch occurs if a single bit in a byte of zero page is clear. This instruction will not be recognised by the BASIC assembler. The assembler opcodes will have to be placed in the program directly using the EQUate instruction. The BRB opcodes are shown below.

BBR Opcodes	
bit 0	&0F
bit 1	&1F
bit 2	&2F
bit 3	&3F
bit 4	&4F
bit 5	&5F
bit 6	&6F
bit 7	&7F

To calculate the offset for relative addressing it should be remembered that as this instruction is 3 bytes long a relative offset of -3 will branch back to the BBR instruction.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	bytes used	cycles
zero page	2	5 (+1 if branch succeeds +2 if to new page)

Example : Branch if bit 1 of &7F=0

```
EQUB &1F     \ BBR opcode for bit 1
EQUB &7F     \ zero page address
EQUB &03     \ branch forward 3 bytes
```

BBS**

Branch on Bit Set Branch if bit=1

BBS is only available on the Rockwell R65C02 microprocessor normally fitted to the Master Turbo or 6502 2nd processors.

The branch occurs if a single bit in a byte of zero page is set. This instruction will not be recognised by the BASIC assembler. The assembler opcodes will have to be placed in the program directly using the EQUate instruction. The BBS opcodes are shown below.

BBS Opcodes	
bit 0	&8F
bit 1	&9F
bit 2	&AF
bit 3	&BF
bit 4	&CF
bit 5	&DF
bit 6	&EF
bit 7	&FF

To calculate the offset for relative addressing it should be remembered that as this instruction is 3 bytes long a relative offset of -3 will branch back to the BBS instruction.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	bytes used	cycles
zero page	2	5 (+1 if branch succeed +2 if to new page)

Example : Branch if bit 1 of &7F=1

```
EQUB &9F      \BBS opcode for bit 1
EQUB &7F      \zero page address (&7F)
EQUB &03      \BBR1,&7F,back -6 bytes
```

BCC

Branch on Carry Clear Branch if C=0

This instruction causes a relative jump if the carry flag is clear. The address to which the branch is directed must be within relative addressing range otherwise the assembler will throw up an "Out of range" message.

Used after a CMP instruction this branch occurs when A<DATA.

Processor Status after use

C (carry flag)	: not affected
Z (zero flag)	: not affected
I (interrupt disable)	: not affected
D (decimal mode flag)	: not affected
B (break command)	: not affected
V (overflow flag)	: not affected
N (negative flag)	: not affected

Addressing mode	Bytes used	Cycles	Op. code
relative	2	2 (+1 if branch succeeds +2 if to new page)	&90

Example : Branch if contents of &80 < 100

```
LDA #100      \ load accumulator with data
CMP &80       \ A-data (comparison)
BCC finish    \ goto finish if ?&80<100
```

BCS

Branch on Carry Set — Branch if C=1

A relative branch will occur if the carry flag is set. The branch address given to the assembler must be within relative addressing range.

Used after a CMP instruction this branch occurs when A>=data.

Processor Status after use

C (carry flag)	: not affected
Z (zero flag)	: not affected
I (interrupt disable)	: not affected
D (decimal mode flag)	: not affected
B (break command)	: not affected
V (overflow flag)	: not affected
N (negative flag)	: not affected

Addressing mode	Bytes used	Cycles	Op. code
relative	2	2 (+1 if branch succeeds +2 if to new page)	&B0

Example : Branch if contents of X register are greater than or equal to 5

```
CPX #5          / X-5, compare
BCS label       / branch to label if X>=5
```

BEQ

Branch on result zero — Branch if Z=1

This instruction causes a relative branch if the zero flag is set when the instruction is executed. The assembler automatically calculates the relative address from the address given and will cause an error if the address is out of range.

Used after a CMP instruction this branch occurs if A=data. Used after an LDA instruction this branch occurs if A=0.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
relative	2	2 (+1 if branch succeeds +2 if to new page)	&F0

Example : Subroutine not used when A=3

```
          CMP #3         \ A-3, comparison
          BEQ over       \ if A=3 goto over
          JSR anything \ subroutine to be missed if A=0

.over     .....
```

BIT

Test memory bits with accumulator — A AND M, N=M7, V=M6

This instruction can be used to test whether one or more specified bits are set. The zero flag is set if the result is 0 otherwise the zero flag is cleared. Bits 7 and 6 of the memory location are transferred to the status register. The BIT instruction performs an AND operation, setting the status flags accordingly, but without storing the result.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if the result=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : set to bit 6 of memory
N (negative flag) : set to bit 7 of memory

When immediate addressing is used (available on the 65C12 only) the V and N flags are not changed.

Addressing mode	Bytes used	Cycles	Op. code
immediate*	2	2	&89
zero page	2	3	&24
absolute	3	4	&2C
absolute,x*	3	4	&3C
zero page,x*	2	4	&34

Example : Test bit 7 of location &8F

```
LDA #&02     \ load mask into accumulator (00000010)
BIT flags    \ A AND flags, if bit 1=1 then Z=0
BNE flag_set \ action to be performed if bit 1 set
```

BMI

Branch if negative flag set Branch if N=1

This relative branch is performed if the result of a previous operation was negative. Relative branch calculations are made by the assembler which will flag an error if an address is given outside the relative addressing range.

A branch occurs after a result which sets bit 7 of the accumulator. (All 8 bit 2's complement negative numbers have this bit set.)

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
relative	2	2 (+1 if branch succeeds +2 if to new page)	&30

Example : Branching if a byte of memory contains a negative number

```
LDA &3010    \ load accum. from memory, N set if -ve
BMI negative \ branch if ?&3010 is negative
```

BNE

Branch on result not zero Branch if Z=0

This instruction causes a relative branch if the zero flag is clear when the instruction is executed. The assembler automatically calculates the relative address from the address given and will cause an error if the address is out of range.

Used after a CMP instruction this branch occurs if A<>data. Used after an LDA instruction this branch occurs if A<>0.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
relative	2	2 (+1 if branch succeeds +2 if to new page)	&D0

Example : Memory location to be written to if it contains zero (i.e. IF ?&84=0 then ?&84=&7F)

```
LDA &84      \ load memory into A to set flags
BNE round    \ if not zero skip the next bit
LDA #&7F     \ load A with value to be written
STA &84      \ write to location &84 .round
....         \ rest of program
```

BPL

Branch on positive result Branch if N=0

Depending on the state of the negative flag a relative branch will be made. The relative address is calculated by the assembler from an address provided by the programmer. This address must be within the relative addressing range.

Branch occurs after a result which sets accumulator bit 7 to 0.

Processor Status after use

C (carry flag)	: not affected
Z (zero flag)	: not affected
I (interrupt disable)	: not affected
D (decimal mode flag)	: not affected
B (break command)	: not affected
V (overflow flag)	: not affected
N (negative flag)	: not affected

Addressing mode	Bytes used	Cycles	Op. code
relative	2	2 (+1 if branch succeeds +2 if to new page)	&10

Example : A loop which shifts A left until bit 7 is set

```
.loop       ASL A          \ shift accumulator 1 bit left
            BPL loop       \ if bit 7 not set then go round again
```

N.B. This will be an endless loop if A=0 on entry.

BRA*

Branch always

The BRA instruction performs a relative branch regardless of the state of the processor state flags. The assembler calculates the relative address from the address given and will cause an error if the address is out of range. This instruction provides a shorter range alternative to the JMP instruction which uses more processor time and when assembled occupies one more byte of program memory.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
relative	2	3 (+1 if to new page)	&80

Example : Branch back to beginning of loop (typewriter program)

```
.loop       JSR getkey    \ get key press value
            JSR printkey \ print key character
            BRA loop      \ loop round again
```

BRK

Forced interrupt **PC and P pushed on stack**
PCL=?&FFFE
PCH=?&FFFF

This instruction forces an interrupt to occur. The processor jumps to the location stored at &FFFE. The program counter is pushed onto the stack followed by the status register. A BRK instruction usually represents an error condition and the BRK handling code is usually an error handling routine. Using machine code in a BASIC environment it is possible to use BASIC's own error handling facilites, see section 2.7. A user BRK handling routine may be implemented, see section 8.13.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	7	&00

Example : Cause an error if A is greater than 4

```
          CMP #5        \ A-5, comparison
          BCC noerr     \ if A<5 then branch round error
          BRK           \ cause error
          ..            \ (error data)
.noerr    ..            \ rest of program
```

BVC

Branch if overflow clear Branch if V=0

A relative branch is made if the overflow flag is clear. The relative address calculation is performed by the assembler which will flag an error if given an address out of relative addressing range.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
relative	2	2 (+1 if branch succeeds +2 if to new page)	&50

Example : Branching on overflow when carry is deliberately set

```
ADC &80          \ A=A+?&80+C
SEC              \ set carry flag
BVC somewhere    \ goto somewhere if no overflow
```

BVS

Branch if overflow set Branch if V=1

Branch to a relative address if the overflow flag is set. Overflow is generally set when the carry flag is set except when a subtraction has been performed. In this case overflow is set when the carry flag is cleared. The address specified in the operand field of the assembler statement must be within the relative addressing range otherwise an assembly error will be flagged.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
relative	2	2 (+1 if branch succeeds +2 if to new page)	&70

Example : Branching if overflow occurs during subtraction

```
SEC          / set the carry flag
LDA #8       / load A with the value 8
SBC &86      / A=A-M (-carry if required)
BVS help     / if overflow has occurred goto help
STA &86      / otherwise put new value in &86
```

N.B. A BCC instruction would have performed the same purpose in this instance.

CLC

Clear carry flag C=0

This instruction clears the carry flag. This is often a sensible operation to perform before using an ADC instruction if there is any doubt as to the status of the carry flag.

Processor Status after use

C (carry flag) : cleared
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&18

Example : Clearing the carry flag before an 8 bit addition

```
CLC             \ clear carry flag
LDA counter     \ load first low order byte
ADC number      \ add second low order to it
STA counter     \ place new value in counter
```

CLD

Clear decimal flag D=0

This flag is used to place the 6502 into decimal mode. This instruction returns the processor into non-decimal mode. The 65C12 microprocessor used in the BBC Master series computers takes one cycle longer than the 6502 to perform decimal mode arithmetic.

See machine code arithmetic, chapter 3.

Processor Status after use

C (carry flag)	: not affected
Z (zero flag)	: not affected
I (interrupt disable)	: not affected
D (decimal mode flag)	: cleared
B (break command)	: not affected
V (overflow flag)	: not affected
N (negative flag)	: not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&D8

Example : Turn decimal mode off

```
        CLD             \ No more BCD arithmetic
```

CLI

Clear interrupt disable flag I=0

This instruction is used to re-enable interrupts after they have been disabled by setting the interrupt flag. In a machine where the operating system relies heavily on interrupts it is unwise to play around with the interrupt flag without good reason. For information about interrupts see chapter 8.

Processor Status after use

C (carry flag)	: not affected
Z (zero flag)	: not affected
I (interrupt disable)	: cleared
D (decimal mode flag)	: not affected
B (break command)	: not affected
V (overflow flag)	: not affected
N (negative flag)	: not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&58

Example : Re-enabled interrupts

```
CLI             \ interrupts responded to now
```

CLR*

Clear memory M=0

This mnemonic is a synonym of the STZ instruction and assembles to produce an op-code which stores a zero at the specified memory location.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
zero page	2	3	&64
zero page,x	2	4	&74
absolute	3	4	&9C
absolute,x	3	5	&9E

Example : clear page 4 of memory

```
            LDX #&FF      \ X=&FF
.loop       CLR &04,X     \ memory location=0
            DEX           \ X=X-1
            BNE loop      \ if not zero, clear next location
            ...           \ rest of program
```

CLV

Clear the overflow flag V=0

This instruction forces the overflow flag to be cleared.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : cleared
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&B8

Example : Explicitly clear the overflow flag

```
CLV             \ overflow now clear
```

CMP

Compare memory and accumulator A-M

This is a very useful instruction for comparing the accumulator contents to the contents of a memory location. The status register flags are set according to the result of a subtraction of the memory contents from the accumulator. The accumulator contents are preserved but the status register flags may be used to cause branches depending on the values which were compared.

Processor Status after use

C (carry flag) : set if A greater than or equal to M
Z (zero flag) : set if A=M
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of the result is set

Addressing mode	Bytes used	Cycles	Op. code
immediate	2	2	&C9
zero page	2	3	&C5
zero page,X	2	4	&D5
(zero page)*	2	5	&D2
absolute	3	4	&CD
absolute,X	3	4 (+1 if page crossed)	&DD
absolute,Y	3	4 (+1 if page crossed)	&D9
(indirect),X	2	6	&C1
(indirect,Y)	2	5 (+1 if page crossed)	&D1

Examples : Branching on the result of a comparison

The test which if true is to cause the branch.			**Code.**

A>M	or	M<A	`BEQ over` (or BEQ P%+4, no label)
			`BCS somewhere`
			`.over ....`

A>=M	or	M<=A	`BCS somewhere`
A=M	or	M=A	`BEQ somewhere`
A<=M	or	M>=A	`BCC somewhere` `BEQ somewhere`
A<M	or	M>A	`BCC somewhere`

CPX

Compare memory with X register X-M

This instruction performs a subtraction of the contents of the memory location from the contents of the X register, the memory location and the register remain intact but the status register flags are set on the result.

Processor Status after use

C (carry flag) : set if X greater than or equal to M
Z (zero flag) : set if X=M
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of the result is set

Addressing mode	Bytes used	Cycles	Op. code
immediate	2	2	&E0
zero page	2	3	&E4
absolute	3	4	&EC

Example : Clearing an area of memory (max &100 bytes). The number of bytes to be cleared is stored in 'count'.

```
            LDA #0          \ set accumulator to 0
            TAX             \ set loop index to 0

.loop       STA page,X      \ write 0 to byte page+X
            INX             \ increment loop index
            CPX count       \ X-?count, comparison
            BNE loop        \ if not equal go round again
```

CPY

Compare memory with Y register Y-M

This instruction subtracts the contents of the specified memory location from the Y register. The memory location and the register remain intact but the status register flags are set on the result.

Processor Status after use

C (carry flag) : set if Y greater than or equal to M
Z (zero flag) : set if Y=M
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of the result is set

Addressing mode	Bytes used	Cycles	Op. code
immediate	2	2	&C0
zero page	2	3	&C4
absolute	3	4	&CC

Example : Branch if Y=&0D

```
CPY #&0D      \ compare Y with &0D/13
BEQ cr        \ if Y=13 goto cr
```

DEC/A*

Decrement memory by one M=M-1

This instruction decrements the value contained in the specified memory location.

The additional immediate addressing mode available on the 65C12 enables the accumulator to be decremented directly. The instruction DEC A may be replaced by the synonym DEA on master series computers.

Processor Status after use

C (carry flag)	: not affected
Z (zero flag)	: set if memory contents become 0
I (interrupt disable)	: not affected
D (decimal mode flag)	: not affected
B (break command)	: not affected
V (overflow flag)	: not affected
N (negative flag)	: set if bit 7 of the result is set

Addressing mode	Bytes used	Cycles	Op. code
accumulator*	1	2	&3A
zero page	2	5	&C6
zero page,X	2	6	&D6
absolute	3	6	&CE
absolute,X	3	7	&DE

Example : Decrement location &2900

```
DEC &2900     \ ?&2900=?&2900-1
```

DEX

Decrement X register by one X=X-1

This instruction decrements the contents of the X register by one.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if X becomes 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of X becomes set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&CA

Example : Decrement X register

```
DEX                 \ X=X-1
```

DEY

Decrement the Y register by one Y=Y-1

This instruction decrements the contents of the Y register by one.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if Y becomes 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of Y becomes set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&88

Example : Decrement Y register

```
DEY                 \ Y=Y-1
```

EOR

Exclusive OR — A=A EOR M

This instruction performs a bit by bit Exclusive OR of the specified memory location contents with the contents of the accumulator leaving the result in the accumulator. The truth table for the logical EOR operation is :-

Acc. bit	Mem. bit	Result bit
0	0	0
0	1	1
1	0	1
1	1	0

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if A becomes 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of A becomes set

Addressing mode	Bytes used	Cycles	Op. code
immediate	2	2	&49
zero page	2	3	&45
zero page,X	2	4	&55
(zero page)*	2	5	&52
absolute	3	4	&4D
absolute,X	3	4 (+1 if page crossed)	&5D
absolute,Y	3	4 (+1 if page crossed)	&59
(indirect,X)	2	6	&41
(indirect),Y	2	5 (+1 if page crossed)	&51

Example : EOR contents of memory with &FF

```
LDA #&FF      \ load accumulator with &FF
EOR temp      \ A=A EOR (?temp)
STA temp      \ reload memory
```

INC/A*

Increment memory by one M=M+1

This instruction increments the value contained in the specified memory location.

The additional immediate addressing mode available on the 65C12 enables the accumulator to be incremented directly. The instruction `INC A` may be replaced by the synonym `INA` on master series computers.

Processor Status after use

C (carry flag)	: not affected
Z (zero flag)	: set if memory contents become 0
I (interrupt disable)	: not affected
D (decimal mode flag)	: not affected
B (break command)	: not affected
V (overflow flag)	: not affected
N (negative flag)	: set if bit 7 of memory becomes set

Addressing mode	Bytes used	Cycles	Op. code
accumulator**	1	2	&1A
zero page	2	5	&E6
zero page,X	2	6	&F6
absolute	3	6	&EE
absolute,X	3	7	&FE

Example : Increment location &80

```
INC &80         \ ?&80=?&80+1
INC A           \ A=A+1
INA             \ A=A+1
```

INX

Increment X register by one X=X+1

This instruction increments the contents of the X register by one.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if X becomes 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of X becomes set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&E8

Example : Increment X register

```
INX             \ X=X+1
```

INY

Increment the Y register by one Y=Y+1

This instruction increments the contents of the Y register by one.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if Y becomes 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of Y becomes set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&C8

Example : Increment Y register

```
INY                 \ Y=Y+1
```

JMP

Jump to new location PC = new address

This instruction is the machine code equivalent of a GOTO statement in BASIC. An indirect addressing mode is available where the address for the JMP is contained in memory specified by the address in the operand field (see examples below).

The 65C12 also allows a pre-indexed absolute indirect addressing mode to be used. Using this addressing mode a jump is made to an address which is determined at run-time by adding the contents of the X register to the contents of the two bytes pointed to by the address field.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
absolute	3	3	&4C
indirect	3	5	&6C
(indirect,X)*	3	6	&7C

Examples : A direct jump

```
JMP entry    \ goto entry
```

An indirect jump (a contrived example)

```
LDA #&00     \ A=0
STA &2800    \ ?&2800=A (address low byte)
LDA #&40     \ A=&40
STA &2801    \ ?&2801=A (address high byte)
JMP (&2800)  \ jump to &4000
```

JSR

Jump Subroutine

Push PC onto stack
PC = new address

This instruction causes a jump but also saves the current program counter on the stack. The subroutine which is called returns to the part of the program that called it by pulling the saved address from the stack and jumping back to the instruction following the JSR. A subroutine must always be terminated by an RTS instruction which performs the return to the location from which the subroutine was called.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
absolute	3	6	&20

Examples : Using an OS call

```
LDA #ASC"X"
JSR OSWRCH   \ print "X" on screen
```

LDA

Load accumulator from memory A=M

This instruction is used to set the contents of the accumulator to that contained in a specified byte of memory.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if A=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of A set

Addressing mode	Bytes used	Cycles	Op. code
immediate	2	2	&A9
zero page	2	3	&A5
zero page,X	2	4	&B5
(zero page)**	2	5	&B2
absolute	3	4	&AD
absolute,X	3	4 (+1 if page crossed)	&BD
absolute,Y	3	4 (+1 if page crossed)	&B9
(indirect,X)	2	6	&A1
(indirect),Y	2	5 (+1 if page crossed)	&B1

Example : Load accumulator with ASCII value for "A"

```
LDA #ASC"A"  \ A=65
```

LDX

Load X register from memory X=M

This instruction is used to set the contents of the X register to that contained in a specified byte of memory.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if X=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of X set

Addressing mode	bytes used	cycles	op. code
immediate	2	2	&A2
zero page	2	3	&A6
zero page,Y	2	4	&B6
absolute	3	4	&AE
absolute,Y	3	4 (+1 if page crossed)	&BE

Example : Load X register with contents of location &80

```
LDX &80          \ X=?&80
```

LDY

Load Y register from memory Y=M

This instruction is used to set the contents of the Y register to that contained in a specified byte of memory.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if Y=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of Y set

Addressing mode	Bytes used	Cycles	Op. code
immediate	2	2	&A0
zero page	2	3	&A4
zero page,X	2	4	&B4
absolute	3	4	&AC
absolute,X	3	4 (+1 if page crossed)	&BC

Example : Load Y register with contents of location labeled 'data' with an offset in X

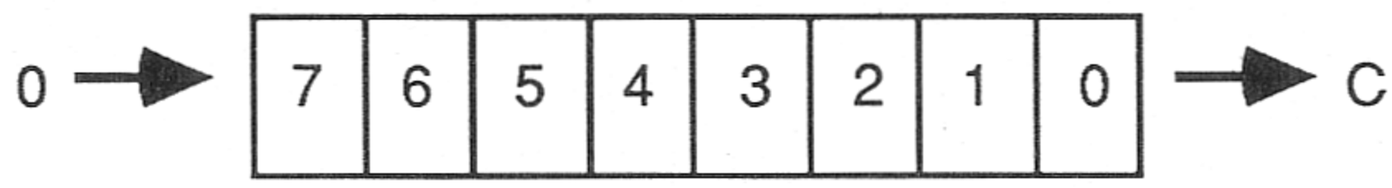
```
LDY data,X   \ Y=?(data+X)
```

LSR

Logical Shift Right by one bit

M=M/2 (or A)

This instruction causes each bit in the memory location or accumulator to shift one bit left. Bit 7 is set to 0 and the carry flag will be set to the old contents of bit 0. The arithmetic effect of this is to divide the value by 2.



Processor Status after use

C (carry flag) : set to bit 0 of operand
Z (zero flag) : set if result=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : cleared

Addressing mode	Bytes used	Cycles	Op. code
accumulator	1	2	&4A
zero page	2	5	&46
zero page,X	2	6	&56
absolute	3	6	&4E
absolute,X	3	7 (+1 if page crossed)	&5E

Example : Shift accumulator contents right one bit

```
        LSR A           \ C=bit 0, A=A/2
```

NOP

No operation

This is a dummy instruction which has no effect on any memory or register contents except to increment the program counter by one.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&EA

Example : A NOP instruction

```
NOP             \ this instruction does nothing
```

ORA

OR memory with accumulator A=A OR M

This instruction performs a bit by bit logical OR operation between the contents of the accumulator and the contents of the specified memory and places the result in the accumulator. The truth table for logical OR is :-

Acc. bit	Mem. bit	Result bit
0	0	0
0	1	1
1	0	1
1	1	1

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if A=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of A set

Addressing mode	Bytes used	Cycles	Op. code
immediate	2	2	&09
zero page	2	3	&05
zero page,X	2	4	&15
(zero page)*	2	5	&12
absolute	3	4	&0D
absolute,X	3	4 (+1 if page crossed)	&1D
absolute,Y	3	4 (+1 if page crossed)	&19
(indirect,X)	2	6	&01
(indirect),Y	2	5 (+1 if page crossed)	&11

Example : Set the top 4 bits of the accumulator

```
ORA #&F0      \ mask is 1111000, 1 OR anything=1
```

PHA

Push accumulator onto stack — Push A

This instruction places the value held in the accumulator onto the stack. This value is accessible using the instruction PLA (pull A from stack).

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	3	&48

Example : Save registers at the beginning of a routine

```
.entry     PHP          / save status register (see below)
           PHA          / save accumulator contents
           TXA          / A=X
           PHA          / save X register contents
           TYA          / A=Y
           PHA          / save Y register contents
            ..          / rest of program
                        /       accumulator not preserved
```

PHP

Push Status register onto stack — Push P

This instruction places the value held in the status register onto the stack. This value is accessible using the instruction PLP (pull P from stack).

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	3	&08

Example : See the example given for PHA above.

PHX*

Push X register onto stack — Push X

This instruction has been implemented on the 65C12 to enable the X register to be placed on the stack directly. On the 6502 where this instruction is not available the X register has to be transferred to the accumulator before it can be saved. The addition of this command helps in two ways. Routines which save the X register on the stack are shorter and faster, as less instructions are required for this operation. It is no longer necessary to make provisions to save the contents of the accumulator if its value needs to be preserved.

Compare the example given below for saving the contents of registers on the stack to the example given with the PHA instruction.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&DA

Example : Save registers at the beginning of a routine

```
.entry    PHP          / save status register (see below)
          PHA          / save accumulator contents
          PHX          / save X register contents
          PHY          / save Y register contents
           ..          / rest of program, all registers preserved
```

PHY*

Push Y register onto stack — Push Y

This instruction has been implemented on the 65C12 to enable the Y register to be placed on the stack directly. On the 6502 where this instruction is not available the Y register has to be transferred to the accumulator before it can be saved. The addition of this command helps in two ways. Routines which save the Y register on the stack are shorter and faster as less instructions are required for this operation. It is no longer necessary to make provisions to save the contents of the accumulator if its value needs to be preserved.

Compare the example given below for saving the contents of registers on the stack to the example given with the PHA instruction.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&5A

Example : Save registers at the beginning of a routine

```
.entry    PHP          / save status register (see below)
          PHA          / save accumulator contents
          PHX          / save X register contents
          PHY          / save Y register contents
           ..          / rest of program, all registers preserved
```

PLA

Pull accumulator off stack — Pull A

This instruction loads the accumulator with a value which is pulled from the stack. This is usually a previous accumulator value which has been saved on the stack using a PHA instruction.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if A=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of A set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	4	&68

Example : Restore registers at the end of a routine

```
PLA             \ pull Y value from stack
TAY             \ put it back in Y
PLA             \ pull X value from stack
TAX             \ put it back in X
PLA             \ pull A value from stack
PLP             \ restore status register
RTS             \ back to calling routine
```

PLP

Pull status register off stack — Pull P

This instruction loads the status register with a value which is pulled from the stack. This is usually a previous status register value which has been saved on the stack using a PHP instruction.

Processor Status after use

C (carry flag) : bit 0 from stack
Z (zero flag) : bit 1 from stack
I (interrupt disable) : bit 2 from stack
D (decimal mode flag) : bit 3 from stack
B (break command) : bit 4 from stack
V (overflow flag) : bit 6 from stack
N (negative flag) : bit 7 from stack

Addressing mode	Bytes used	Cycles	Op. code
implied	1	4	&28

Example : Restore registers at the end of a routine

```
PLY             \ pull Y value from stack
PLX             \ pull X value from stack
PLA             \ pull A value from stack
PLP             \ restore status register
RTS             \ back to calling routine
```

PLX*

Pull X from stack — Pull X

This instruction has been implemented on the 65C12 to enable the X register to be removed from the stack directly. On the 6502 where this instruction is not available the X register value on the stack has to be transferred to the accumulator before it can be restored to the register. The addition of this command helps in two ways. Routines which restore the X register from the stack are shorter and faster as less instructions are required for this operation. It is no longer necessary to make provisions to save the contents of the accumulator if its value needs to be preserved.

Compare the example given below for restoring the contents of registers from the stack to the example given with the PLA instruction.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if X=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of X set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&FA

Example : Restore registers at the end of a routine

```
PLY             \ pull Y value from stack
PLX             \ pull X value from stack
PLA             \ pull A value from stack
PLP             \ restore status register
RTS             \ back to calling routine
```

PLY*

Pull Y from stack — Pull Y

This instruction has been implemented on the 65C12 to enable the Y register to be removed from the stack directly. On the 6502 where this instruction is not available the Y register value on the stack has to be transferred to the accumulator before it can be restored to the register. The addition of this command helps in two ways. Routines which restore the Y register from the stack are shorter and faster as less instructions are required for this operation. It is no longer necessary to make provisions to save the contents of the accumulator if its value needs to be preserved.

Compare the example given below for restoring the contents of registers from the stack to the example given with the PLA instruction.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if Y=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of Y set

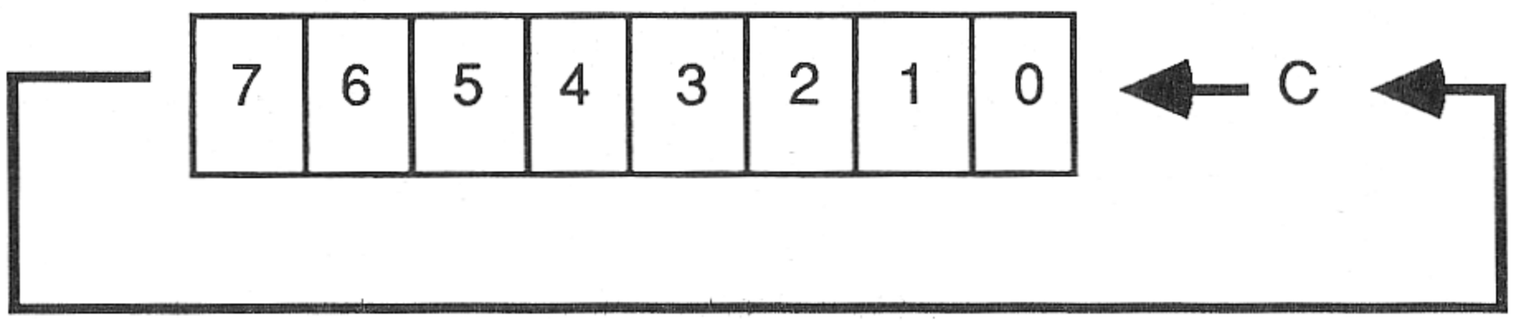
Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&7A

Example : See example for PLX instruction.

ROL

Rotate one bit left — M=M*2, M0=C, C=M7 (A or M)

This instruction causes a shift left one bit. The bit shifted out of the byte, bit 7, is placed in the carry flag. The contents of the carry flag are placed in bit 0.



Processor Status after use

C (carry flag) : set to old value of bit 7
Z (zero flag) : set if result=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of the result is set

Addressing mode	Bytes used	Cycles	Op. code
accumulator	1	2	&2A
zero page	2	5	&26
zero page,X	2	6	&36
absolute	3	6	&2E
absolute,X	3	7	&3E

Example : Rotate accumulator contents one bit left

```
ROL A          \ A=A rotated left
```

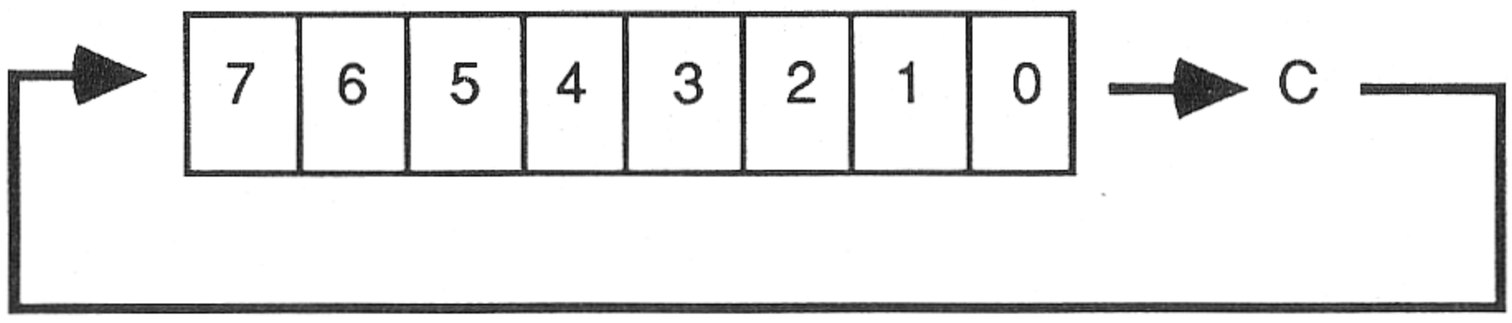
N.B. The carry flag state should be known before this operation is performed.

ROR

Rotate one bit right

M=M/2, M7=C,
C=M0 (A or M)

This instruction causes a shift right one bit. The bit shifted out of the location, bit 0 is placed in the carry flag. The contents of the carry flag are placed in bit 7.



Processor Status after use

C (carry flag) : set to old value of bit 0
Z (zero flag) : set if result=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of the result is set

Addressing mode	Bytes used	Cycles	Op. code
accumulator	1	2	&6A
zero page	2	5	&66
zero page,X	2	6	&76
absolute	3	6	&6E
absolute,X	3	7	&7E

Example : Reverse the order of bits in a byte

```
.start      STA &80       \ store byte in &80
            LDX #8        \ set loop count to 8

.loop       ROL &80       \ bit 7 of &80 to carry
            ROR A         \ carry to bit 8 of A
            DEX           \ decrement loop count
            BNE loop      \ if not 0 goto loop
            RTS           \ exit with A reversed
```

RTI

Return from Interrupt

Status register and PC pulled from stack

This instruction is used to return from an interrupt handling routine. When an interrupt occurs the current program counter and status register are pushed onto the stack. These are restored by the RTI instruction.

Processor Status after use

C (carry flag) : bit 0 from stack
Z (zero flag) : bit 1 from stack
I (interrupt disable) : bit 2 from stack
D (decimal mode flag) : bit 3 from stack
B (break command) : bit 4 from stack
V (overflow flag) : bit 6 from stack
N (negative flag) : bit 7 from stack

Addressing mode	Bytes used	Cycles	Op. code
implied	1	6	&40

Example : Instruction at the end of an interrupt handling routine

```
....          \ code dealing with the interrupt
RTI           \ back to what we were doing before.
```

RTS

Return from subroutine Pull PC from stack

The RTS instruction is used to terminate the execution of a subroutine. Any routine terminated in this way should be called using a JSR instruction which places a return address on the stack. The top two stack values are placed in the program counter and execution is resumed at the point in the program after the JSR instruction. During a subroutine the same number of items pushed on the stack must be removed before the RTS instruction is reached if the subroutine is to return to the correct address.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	6	&60

Example : Last instruction in a subroutine

```
          ....          \ body of subroutine
        RTS             \ return to calling routine
```

SBC

Subtract memory from accumulator with carry

A,C=A-M-(1-C)

This instruction subtracts the contents of the specified memory from the accumulator contents leaving the result in the accumulator. If the carry flag is used as a 'borrow' source and if clear then an extra unit is subtracted from the accumulator. This enables the 'borrow' to be carried over in multi-byte subtractions (see example below).

Processor Status after use

C (carry flag) : cleared if a borrow occurs
Z (zero flag) : set if result=0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : set if the sign of the result is wrong
N (negative flag) : set if bit 7 of the result is set

Addressing mode	Bytes used	Cycles	Op. code
immediate	2	2	&E9
zero page	2	3	&E5
zero page,X	2	4	&F5
(zero page)*	2	5	&F2
absolute	3	4	&ED
absolute,X	3	4 (+1 if page crossed)	&FD
absolute,Y	3	4 (+1 if page crossed)	&F9
(indirect,X)	2	6	&E1
(indirect),Y	2	5 (+1 if page crossed)	&F1

Example : 16 bit value at locations &80 and &81 subtracted from 16 bit value at locations &82 and &83, result at locations &82 and &83.

```
SEC             \ ready for any 'borrow'
LDA &80         \ low order byte of first value
SBC &82         \ A=A-?&82 (borrow may occur)
STA &82         \ place result in &82
LDA &81         \ high order byte of first value
SBC &83         \ A=A-&83-(-C)
STA &83         \ place result in &83
```

SEC

Set carry flag C=1

This instruction is used to set the carry flag. It should be used to set the carry flag prior to a subtraction unless the carry flag has been deliberately left as a 'borrow' from a previous subtraction.

Processor Status after use

C (carry flag) : set
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&38

Example : Explicit setting of the carry flag

```
SEC             / C=1
```

SED

Set decimal mode D=1

This instruction is used to place the 6502 in decimal mode. It causes arithmetic operations to be performed in BCD mode.

Note that the 65C12 as used on the master series computers uses an additional clock cycle to perform arithmetic instructions in decimal mode compared with the 6502.

See machine code arithmetic, chapter 3.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : set
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&F8

Example : Set decimal mode for arithmetic

```
SED             \ BCD from now on
```

SEI

Set interrupt disable flag I=1

This instruction is used to set the interrupt disable flag. When this flag is set maskable interrupts cannot occur. See interrupts chapter 8.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : set
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&78

Example : Disable interrupts

```
SEI             \ No maskable interrupts
```

STA

Store accumulator contents in memory M=A

This instruction is used to copy the contents of the accumulator into a memory location specified in the operand field.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
zero page	2	3	&85
zero page,X	2	4	&95
(zero page)*	2	6	&92
absolute	3	4	&8D
absolute,X	3	5	&9D
absolute,Y	3	5	&99
(indirect,X)	2	6	&81
(indirect),Y	2	6	&91

Example : Store accumulator in location 'save' + Y offset

```
STA save,Y   \ ?(save+Y)=A
```

STX

Store X contents in memory M=X

This instruction is used to copy the contents of the X register into a memory location.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
zero page	2	3	&86
zero page,Y	2	4	&96
absolute	3	4	&8E

Example : Store X in location &80

```
STX &80        \ ?&80=X
```

STY

Store Y contents in memory M=Y

This instruction is used to copy the contents of the Y register into a memory location.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
zero page	2	3	&84
zero page,X	2	4	&94
absolute	3	4	&8C

Example : Store Y in location &5FF0

```
STY &5FF0     \ ?&5FF0=Y
```

STZ*

Clear memory M=0

This mnemonic is a synonym of the CLR instruction and assembles to produce an op-code which stores a zero at the specified memory location.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
zero page	2	3	&64
zero page,x	2	4	&74
absolute	3	4	&9C
absolute,x	3	5	&9E

Example : clear page 4 of memory

```
          LDX #&FF      \ X=&FF
.loop     STZ &04,X     \ memory location=0
          DEX           \ X=X-1
          BNE loop      \ if not zero, clear next location
          ...           \ rest of program
```

TAX

Transfer A to X X=A

This instruction is used to copy the contents of the accumulator to the X register.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if X becomes 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of X is set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&AA

Example : Transfer contents of A to X

```
TAX             \ X=A
```

TAY

Transfer A to Y Y=A

This instruction is used to copy the contents of the accumulator to the Y register.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if Y becomes 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of Y is set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	A8

Example : Transfer contents of A to Y

```
TAY             \ Y=A
```

TRB*

Test and reset bits

This instruction takes the complement of the accumulator and performs a logical AND with the contents of the byte of memory determined by the address field. The result is placed in the memory location.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if A AND memory = 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
zero page	2	5	&14
absolute	3	6	&1C

TSB*

Test and set bits

This instruction takes the accumulator value and performs a logical OR with the contents of the byte of memory determined by the address field. The result is placed in the memory location.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if A AND memory = 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
zero page	2	5	&04
absolute	3	6	&0C

TSX

Transfer S to X X=S

This instruction is used to copy the contents of the stack pointer to the X register.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if X becomes 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of X is set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&BA

Example : Transfer contents of S to X

```
TSX       \ X=S
```

TXA

Transfer X to A

A=X

This instruction is used to copy the contents of the X register to the accumulator.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if A becomes 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of A is set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&8A

Example : Transfer contents of X to A

```
TXA             \ A=X
```

TXS

Transfer X to S S=X

This instruction is used to copy the contents of the X register to the stack pointer.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : not affected
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : not affected

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&9A

Example : Transfer contents of X to S

```
TXS             \ S=X
```

TYA

Transfer Y to A A=Y

This instruction is used to copy the contents of the Y register to the accumulator.

Processor Status after use

C (carry flag) : not affected
Z (zero flag) : set if A becomes 0
I (interrupt disable) : not affected
D (decimal mode flag) : not affected
B (break command) : not affected
V (overflow flag) : not affected
N (negative flag) : set if bit 7 of A is set

Addressing mode	Bytes used	Cycles	Op. code
implied	1	2	&98

Example : Transfer contents of Y to A

```
        TYA             \ A=Y
```

6 Introduction to the OS

The operating system is a complex piece of software designed to provide a consistent and comprehensive range of routines servicing the computer hardware.

Calls to the operating system allow characters to be sent to the screen, key strokes to be read from the keyboard, data to be down the serial port, etc. etc.

The operating system software is programmed into ROMs which occupy the top 16 Kb of the 6502 memory map from &C000 to &FFFF (bar a gap for memory mapped i/o). In any of the example programs in this book, calls are made to the operating system by calling sub-routines within this memory. A typical call to write the letter 'A' to the screen might look like this.

```
LDA #65       \ ASCII code for 'A' is 65
JSR &FFEE     \ OS write character routine
```

Many operating system routines make use of vectors stored in RAM. These vectors are words of RAM (two-bytes) reserved for storing the address of a particular routine. When a vectored OS call is used, the first action of the operating system routine is to jump to the address held in the vector. Thus the code in the OS write character routine used in the example above looks like this :

```
JMP (&020E)  \ &20E + &20F contain the address &E822
```

(note that &E822 is the address placed in the vector by the Master OS, different values will be found in this vector on other microcomputers in the Acorn BBC series)

The non-vectored write character routine in contrast looks like this:

```
JMP &E822    \ the body of the routine is at &E822
```

Clearly the non-vectored write character routine will take less time to execute but the existence of a vector stored in RAM enables the interception of the operating system routine by a different routine. This is of particular benefit where different filing systems may be used for floppy disc drives, tape recorders or ROM cartridges. As each filing system is selected, the filing system vectors may be altered to point at the relevant filing system software. Thus programs which use the filing system facilities will work similarly no matter which filing system is selected. The operating system calls which load and save files will remain unchanged but the actual body of the routines which perform the work will depend on the address stored in the vector.

The addresses of the vectors used by each of the operating system routines are given in the sections below. Other vectors used to provide to allow access to facilities such as interrupts or buffer handling are described in the appropriate chapters.

A well designed operating system consists of a limited number of sub-routines, the actions of which are determined by parameters passed to those sub-routines. The format of data passed to the operating system calls should be consistent, as should the behaviour of the calls and the return of data.

Subsequent versions of an operating system should provide the same facilities as previous operating systems while allowing the use of new facilities using the same type of calls. Good initial design allows for continuous improvement and expansion with little disruption to the original design. Applications software which uses operating system calls correctly should continue to run with later operating system versions.

6.1 Operating System Calls

OS Routine		OS Vector		Summary of function
Name	Address	Name	Address	
		USERV	&200	the User vector
		BRKV	&202	the BRK vector
		1RQ1V	&204	Primary interrupt vector
		1RQ2V	&206	Unrecognised IRQ vector
OSCLI	&FFF7	CLIV	&208	Command line interpreter
OSBYTE	&FFF4	BYTEV	&20A	*FX/OSBYTE call
OSWORD	&FFF1	WORDV	&20C	OSWORD call
OSWRCH	&FFEE	WRCHV	&20E	Write character to output stream
OSNEWL	&FFE7	-	-	Write LF,CR to screen
OSASCI	&FFE3	-	-	Write character, &D gives CR and LF
OSRDCH	&FFE0	RDCHV	&210	Get character from input stream
OSFILE	&FFDD	FILEV	&212	Load/Save file
OSARGS	&FFDA	ARGSV	&214	Load/Save file parameters
OSBGET	&FFD7	BGETV	&216	Get byte from file
OSBPUT	&FFD4	BPUTV	&218	Put byte to file
OSGBPB	&FFD1	GBPBV	&21A	Multiple BPUT/BGET
OSFIND	&FFCE	FINDV	&21C	Open/Close file
		FSCV	&21E	Filing system control entry vector
		EVNTV	&220	Event vector
		UPTV	&222	User print vector
		NETV	&224	Econet vector
		VDUV	&226	Unrecognised VDU command vector
		KEYV	&228	Keyboard vector
		INSV	&22A	Insert character into buffer vector
		REMV	&22C	Remove character from buffer vector
		CNPV	&22E	Count/Purge buffer vector
		IND1V	&230	Spare vector
		IND2V	&232	Spare vector
		IND3V	&234	Spare vector
NVWRCH	&FFCB	-	-	Non-vectored read character
NVRDCH	&FFC8	-	-	Non-vectored write character
GSREAD	&FFC5	-	-	Read character from string
GSINIT	&FFC2	-	-	String input initialise
OSEVEN	&FFBF	-	-	Generate an event
OSRDSC	&FFB9	-	-	Read byte from screen or 'paged ROM'
OSWRSC	&FFB3	-	-	Write byte to screen or 'paged ROM'

All operating system calls take their primary parameter in the accumulator. Secondary parameters, when required are passed in the X and Y registers. If more data is required for the execution of an operating system call then this data is stored in a block of memory known as a *parameter block*, the address of which is passed in the X and Y registers (least significant byte of the address in X, most significant in Y).

The following sections contain a brief description of the general function of each operating system call. Where appropriate detailed call descriptions are contained in chapters covering the relevant system. So OSBYTE calls dealing with the video system are described in chapter 13 while the general output calls are included below.

The operating system calls may be divided into four broad functional groups:

I/0 routines
OSBYTE/OSWORD
Filing system routines
miscellaneous functions

6.2 I/O routines

The operating system calls which implement input/output facilities are described below. A description of the OS calls concerned with the i/o buffers is included in chapter 9. Keyboard specific calls, including the function keys and cursor editing control, are described in chapter 14. Printer handling routines are covered in chapter 24.

6.2.1 OSWRCH : Write character routine

Call address &FFEE
Indirected through &20E

Implemented on all Acorn BBC microcomputers.

This routine outputs the character in the accumulator to the currently selected input stream(s).

While the indirection vector may be used in a 6502 second processor, it should be noted that OSWRCH calls generated in the i/o processor are not passed across the tube e.g. filing system messages.

On exit:

A, X and Y are preserved.
C, N, V and Z are undefined.
The interrupt status is preserved (though interrupts may be enabled during a call).

6.2.2 Non-vectored OSWRCH

Call address &FFC8

This routine is implemented on all Acorn BBC microcomputers but is not available to software running on a 6502 second processor.

This call uses the same code as the default OSWRCH routine. It has no indirection vector and its general use is not recommended.

6.2.3 OSNEWL Write a new-line routine

Call address &FFE7
Not indirected

This call is implemented on all Acorn BBC microcomputers.

This routine writes a carriage return (&D/13) and a line feed (&A/10) to the current output stream(s) using OSWRCH.

On exit:
A=&0D (13)
X and Y are preserved.
C, N, V and Z are undefined.

Interrupt status is preserved (though it may be enabled during a call).

6.2.4 OSASCI Write character routine, OSNEWL called if A=&0D (13).

Call address &FFE3
Not indirected

This call is implemented on all Acorn BBC microcomputers.

This is a write character routine which outputs a carriage return and a line feed if passed a carriage return character (&D/13) in the accumulator.

On exit:
A, X and Y are preserved.
C, N, V and Z are undefined.
Interrupt status is preserved (though interrupts may be enabled during a call).

Information about the output effects resulting from the character values passed to these OS routines is given in the Video chapter (13).

6.2.5 Selecting the input stream

Select input stream OSBYTE call

In the Electron any call with X<>0 will result in and unknown OSBYTE service call being made to the paged ROMs unless a previous such was recognised and thus changed the input source.

Call address &FFF4
Indirected through &20A
Entry parameters:
A=&02
X determines input source(s)

X=0 keyboard selected, RS423 disabled (*FX2,0)
X=1 RS423 selected and enabled (*FX2,1)
X=2 keyboard selected, RS423 enabled (*FX2,2)

Default: X=1

Read input source flag OSBYTE call

A=&B1 (177)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X)

This flag value should be 0 for keyboard input and 1 for RS423 input (i.e. contains buffer number) and is used by the OSBYTE above. This call should not be used to alter the input source as writing the flag does not enable the relevant interrupts.

6.2.6 OSRDCH Read character routine

Call address &FFE0
Indirected through &210

Implemented on all Acorn BBC microcomputer operating systems.

This routine reads a character from the currently selected input stream and returns it in the accumulator.

While the indirection vector may be used in a 6502 second processor, it should be noted that OSRDCH calls generated in the i/o processor are not passed across the tube.

On exit:

C=0 indicates that a valid character has been read.
C=1 indicates that a character has not been read due to a error.

If an error should occur acknowledgement of the error condition should be made.

X and Y are preserved.
N, V and Z are undefined.

The interrupt status is preserved (though interrupts may be enabled during a call).

6.2.7 Non-vectored OSRDCH

Call address &FFC8

This routine is implemented on all Acorn BBC microcomputers but is not available to software running on a 6502 second processor.

This call uses the same code as the default OSRDCH routine and has no indirection vector. Its general use is not recommended.

6.2.8 Read line from input OSWORD call

Call address &FFF1
Indirected through &20C

On entry,

A=0
X contains low byte of the parameter block address.
Y contains high byte of the parameter block address.

This routine takes a specified number of characters from the currently selected input stream. Input is terminated following a RETURN or an ESCAPE. DELETE (&7F/127) deletes the previous character and CTRL U (&15/21) deletes the entire line. If characters are presented after the maximum line length has been reached, the extra characters are ignored and a BEL (ASCII 7) character is output.

The parameter block :-

XY +	0	Buffer address for input - LSB
	1	Buffer address for input - MSB
	2	Maximum line length
	3	Min. acceptable character value
	4	Max acceptable character value

Only characters greater or equal to XY+3 and less than or equal to XY+4 will be accepted.

On exit:

C=0 if a carriage return terminated input.
C=1 if an ESCAPE condition terminated input.
Y contains line length, including carriage return if used.

6.2.9 GSINIT — General string input initialise routine.

Call address &FFC2

This routine is implemented on all the Acorn BBC microcomputers but is not available in the Tube operating system.

This routine initialises a string for input prior to reading using GSREAD.

Entry parameters:

String address stored in &F2 and &F3 plus offset in Y

On exit:

Y contains the offset of the first non-blank character from the address contained in &F2 and &F3.

A contains the first non-blank character of the string

Z flag is set if the string is a null string

6.2.10 GSREAD — Read character from string input routine.

Call address &FFC5

This routine is used to read characters from an input string after a GSINIT call. Control codes and non-ASCII values may be introduced by using an escape character, '|'. The escape character followed by a letter gives the ASCII value minus 64 (&40). The escape character followed by a '!' character gives a value of 128 (&80) plus the value of the following character. An escape character followed by itself gives the escape character.

Entry parameters:

&F2, &F3 and Y set by GSINIT

C=0 String terminated by first space, carriage return or second quotation mark.
C=1 String terminated by carriage return or second quotation mark.

On exit:

A contains the character read from the string.
Y contains the index for the next character to be read.
C=1 if the end of string is reached.
X is preserved.

6.2.11 Selecting the output stream

Select output stream OSBYTE call

Call address &FFF4
Indirected through &20A

If RS423 output is selected in the Electron, paged ROM service calls are issued and in the absence of a suitable response this output is sunk. The same applies to printer output if selected.

Bit 3 should not be used to enable the printer as this may conflict with the Econet protocol of claiming the printer.

In the BBC OS 1.20, control characters will be passed to the printer (even when disabled) when the VDU is disabled.

Entry parameters:

A=3

X determines output device(s)

Y=0

X bit	o/p selected if bit is set
0	Enables RS423 driver
1	Disables VDU driver
2	Disables printer driver
3	Enables printer, independent of CTRL B or C
4	Disables spooled output
5	Not used
6	Disables printer driver unless the character is preceded by a VDU 1 (or equivalent)
7	Not used

*FX 3,0 selects the default output options which are :

RS423 disabled

VDU enabled

Printer enabled (if selected by VDU 2)

Spooled output enabled (if selected by *SPOOL)

On exit:

A is preserved

X contains the old output stream status

Y and C are undefined

Read/write output stream flag OSBYTE call

A=&EC (236)Read/write output stream flag OSBYTE call

Call address &FFF4
Indirected through &20A

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X)

This call may be used to access the flag used by the OSBYTE call described above.

6.3 OSBYTE & OSWORD calls

These two operating system calls are the real work horses of the operating system. They provide a standard protocol for controlling virtually all hardware facilities.

The principal parameter of both calls is the eight bit value passed in the accumulator. This value determines the action of the call allowing a

total of 256 variants of each call. OSBYTE and OSWORD calls are identified by the value passed in the accumulator. For example the OSBYTE call which selects serial or parallel printer output is performed by calling &FFF4, the address of the OSBYTE routine, with an accumulator value of 5. It is generally referred to as OSBYTE &05. The OSBYTE calls are also accessible using the *FX command via the command line interpreter. Selection of the printer port using this method can be achieved by typing *FX5,1 or *FX5,2

6.3.1 OSBYTE call

Call address &FFF4
Indirected through &20A

This is an operating system routine performing a wide range of tasks. Three eight bit parameters are provided in the A,X and Y registers. Any results are returned in the X and Y registers. Some OSBYTE calls also return significant information via the carry flag of the status register.

Entry parameters:

A selects the OSBYTE routine
X contains the first parameter
Y contains the second parameter

On exit:

A is preserved
X may contain the first returned value
Y may contain the second returned value

OSBYTE calls may also be executed by passing the string "FX,x,y" to the command line interpreter where the first decimal value 'x' is placed in the X register and the second decimal value 'y' is placed in the Y register. No values can be returned from OSBYTE calls executed in this manner.

OSBYTE calls made with accumulator values between &A6(166) and &FF(255) are used to read or write operating system status flags or variables. The action of these calls is to replace the contents of the specified location with :

(<OLD VALUE> AND Y) EOR X

To read a location set X=0 and Y=&FF

To write a location set X=<value> and Y=0

Many of these calls repeat the function of lower value OSBYTEs (these equivalent calls are not guarantied to have an identical effect when used

to set flags or OS variables because other actions may also be performed by the lower value OSBYTE). Some OSBYTE calls have little or no practical value and are included for completeness.

OSBYTE calls unrecognised by the operating system are offered to the paged ROMS using a service call of value &07 with the A, X and Y register values stored in zero page locations &EF, &F0 and &F1 respectively.

6.3.2 OSWORD call

Call address &FFF1
Indirected through &20C

The OSWORD call performs a wide range of tasks. Different OSWORD routines are selected by calling the routine with different values in the accumulator. The X and Y registers contain the address of a parameter block which may be used to pass an unlimited number of parameters to the routine. Parameters are returned in a block of memory at the same address.

Entry parameters:
X contains least significant byte of the parameter block address
Y contains most significant byte of the parameter block address

Exit parameters:
Placed in memory at the address specified by the entry parameters

Unknown OSWORD calls in the range &E0 - &FF are passed to the user vector (USERV, &200). Other unknown OSWORD calls are passed to the paged ROMs using a paged ROM service call with A=&08. The A, X and Y register values are stored in zero page locations &EF, &F0 and &F1 respectively.

A word of warning to any programmer intercepting unknown OSWORDs greater than &80 in paged ROMs. There is a problem in machines containing DFS 0.90. This version of DFS (found in BBC model B micro's only) intercepts any unknown OSWORD in this region and assumes it is OSBYTE &7F.

6.4 Filing System Calls

The following calls are used to provide filing system utilities:

OSFILE	&FFDD	(FILEV	&212)
OSARGS	&FFDA	(ARGSV	&214)
OSBGET	&FFD7	(BGETV	&216)
OSBPUT	&FFD4	(BPUTV	&218)
OSGBPB	&FFD1	(GBPBV	&21A)
OSFIND	&FFCE	(FINDV	&21C)
Filing system control entry		(FSCV	&21E)

These calls are described in the filing systems chapter.

6.5 Miscellaneous OS calls

These miscellaneous functions are performed by the calls described below.

Entry parameters:

X and Y contain the address of the command string (X=LSB)
(the command string must be terminated by a carriage return)

On exit:

All registers and status flags are undefined.
(return to the calling routine may not occur in all cases)

6.5.1 OSCLI

Call address &FFF7
Indirected through &208 (CLIV)

OSCLI passes a string of text to the command line interpreter (CLI). The CLI is routine which interprets the *commands and acts accordingly. Normally *commands are typed at the keyboard as part of a BASIC command. On the Master series computer the CLI can be entered directly following a *GO command (with no parameters) or by disabling all language ROMs. This routine provides a method of executing *commands from machine code. The effects are the same as if the command had been issued from the keyboard.

6.5.1 OSRDSC

Call address &FFB9
Not indirected

This call, previously christened OSRDRM in the original 'Advanced User Guide' is used to read the screen or paged ROM memory. This call is not available from second processors.

Entry parameters:

Least significant byte of the address to be read in location &F6
Most significant byte of the address to be read in location &F7
Y= ROM number

On Exit:

A contains the byte read.

6.5.2 OSWRSC

Call address &FFB3
Not indirected

Entry parameters:

Least significant byte of the address to be read in location &D6
Most significant byte of the address to be read in location &D7
A=value to be written

On Exit:

A, X and Y are preserved.

This call is not available from second processors.

6.5.3 OSEVEN

Call address &FFBF
Not indirected

This call is used to generate an event subject to that event having been enabled using OSBYTE &0E (14). Events are described in chapter 7.

Entry parameters:

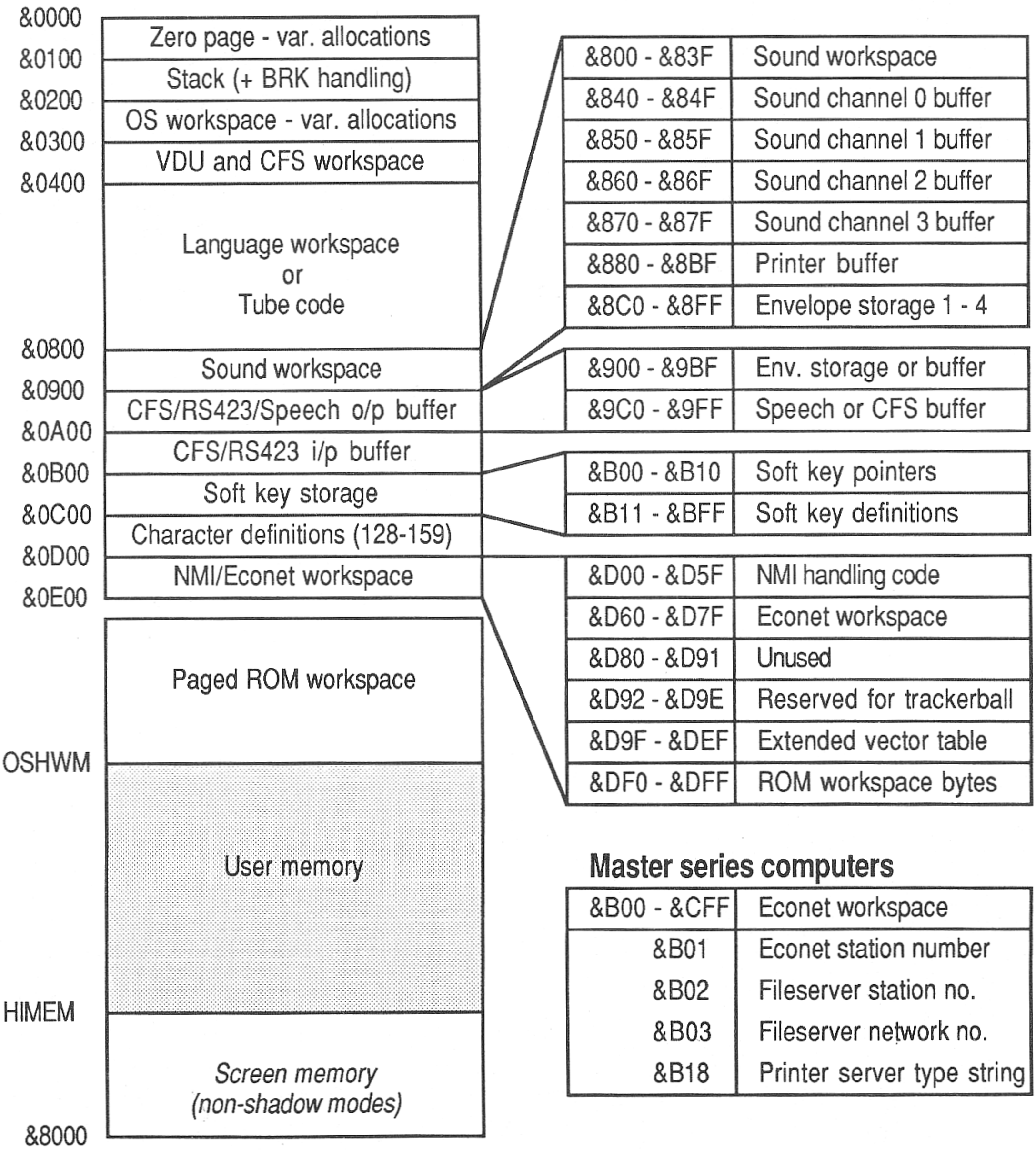
Y=event number

EVNTV is entered with
X is preserved
A=event number
Y=original accumulator value.

This call is not implemented on second processors.

6.6 OS allocation and use of memory

The following diagram illustrates the operating system's use and allocation of RAM.



Master series computers

&B00 - &CFF	Econet workspace
&B01	Econet station number
&B02	Fileserver station no.
&B03	Fileserver network no.
&B18	Printer server type string

6.6.1 Page zero allocations

Zero page memory is a valuable commodity in a 6502 based microcomputer as a number of useful addressing modes can only use

this page of memory. Zero page has been allocated according to the scheme described in the following diagram.

add.	memory usage
&00 &8F	Language work space
&90 &9F	Econet work space
&A0 &A7	NMI work space
&A8 &AF	OS temp work space
&B0 &BF	FS temp work space
&C0 &CF	Filing system work space
&D0 &E1	VDU work space
&E2	CFS/RFS status byte
&E3	CFS/RFS options byte
&E4 &E6	General OS work space
&E7	Auto repeat countdown byte
&E8 &E9	OSWORD &00 input pointer
&EA	Serial timeout counter
&EB	CFS/RFS 'critical' flag
&EC	Last key press (int. key. no.)
&ED	Penultimate key press (int. key no.)
&EE	1MHz paging register
&EF	OSBYTE/OSWORD Accumulator val.
&F0	OSBYTE/OSWORD X register value
&F1	OSBYTE/OSWORD Y register value
&F2 &F3	Command line pointer (with Y register)
&F4	Copy of ROM select register
&F5	Speech PHROM/RFS ROM number
&F6 &F7	PHROM/RFS ROM pointer
&F8 &F9	Soft key expansion ptr (Compact only)
&FA &FB	General OS workspace
&FC	Accumulator storage during interrupts
&FD &FE	Program counter after last BRK
&FF	Escape flag (bit 7 only)

&70 &8F	User zero page (BASIC)

&D0	STATS - VDU status byte
&D1	ZMASK - current graphics point mask
&D2	ZORA - text colour OR mask
&D3	ZEOR - text colour EOR mask
&D4	Graphics colour OR mask
&D5	Graphics colour EOR mask
&D6 &D7	Graphics character cell address
&D8 &D9	Text character cell address
&DA &DB &DC &DD &DE &DF	temporary VDU work space ZTEMP, ZTEMPA, ZTEMPB
&E0 &E1	Ptr to mult. tables (not Master series)

6.6.2 General OS workspace &200-&2FF

Page two of memory is used for a number of miscellaneous operating system functions. The table below contains much information which is unofficial and should therfore be used with caution.

add.	memory usage
&200 &235	Operating system vectors
&236 &28F	OS variables (OSBYTEs &A6 - &FF)
&290	*TV vert. adjust (OSBYTE &90)
&291	*TV interlace setting (OSBYTE &90)
&292 &296	TIME value - first copy
&297 &29B	TIME value - second copy
&29C &2A0	Interval timer value (OSWORD 3+4)
&2A1 &2B0	Paged ROM table (OSBs &AA/AB)
&2B1 &2B2	INKEY countdown counter
&2B3 &2B5	OSWORD &00 work space
&2B6 &2B9	LSBs of most recent ADC conversions
&2BA &2BD	MSBs of most recent ADC conversions
&2BE	Last ADC channel to finish conversion
&2BF &2C8	Event enable flags (OSBYTE &D/&E)
&2C9	Soft key exp. ptr. (not Master series)
&2CA	Auto repeat count for next key
&2CB &2CD	Two key rollover work space
&2CE	Sound semaphore
&2CF &2D7	Buffer busy flags
&2D8 &2E0	Buffer offset - current removal pointer
&2E1 &2E9	Buffer offset - current insertion pointer
&2EA &2EB	CFS open i/p file current block size
&2EC	CFS open i/p file current block flag
&2ED	CFS i/p file last char. of current block
&2EE &2FF	OSFILE ctrl. blocks for *LOAD/SAVE

Differences in Master series computers

&2C9	Auto repeat count for next key
&2CA &2CC	Two key rollover work space
&2CD	Sound semaphore
&2CE &2D6	Buffer busy flags
&2D7 &2DF	Buffer offset - current removal pointer
&2E0 &2E8	Buffer offset - current insertion pointer
&2E9 &2EA	CFS open i/p file current block size
&2EB	CFS open i/p file current block flag
&2EC	CFS i/p file last char. of current block
&2ED	Unused byte

6.6.3 OS memory OSBYTE calls

Read top of operating system RAM address (OSHWM)

Call address &FFF4
Indirected through &20A
A=&83 (131)

This call returns the address of the first byte of main memory that has not already been allocated by the operating system for its own use or allocated for use by paged ROMs. This value is called the *operating system high-water mark* (OSHWM). The low byte of the address is returned in X, the high byte in Y.

Read or write OSHWM

Call address &FFF4
Indirected through &20A
A=&B4 (179)

This call may be used to read or modify the value which the operating system uses as OSHWM. This value is the MSB of a 16 bit address which is always on a page boundary.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

Read or write primary OSHWM

Call address &FFF4
Indirected through &20A
A=&B3 (178)

The primary OSHWM is a value representing the top of operating system memory regardless of the state of the user font definitions. On the Master series computers, where the state of the font does not affect this value, this OSBYTE is used to access the paged ROM 100Hz polling semaphore.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

For other OSBYTE calls returning information about memory settings see chapter 12.

Read or write OS call interception status OSBYTEs

Call address &FFF4
Indirected through &20A
A=&CE (206)A=&CF (207)
A=&D0 (208)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

These three calls return the bytes which are used as flags to indicate which operating system calls have been intercepted by the NFS. The top bit of each byte is used as the flag. If bit 7 is set, then all calls to the relevant routines are routed through the Econet vector (&224). OSBYTE &CE returns the status of the OSBYTE and OSWORD vectors, OSBYTE &CF returns the status of the OSRDCH vector and &D0 returns the status of the OSWRCH vector.

Read or write address of OS variables

Call address &FFF4
Indirected through &20A
A=&A6 (166)A=&A7 (167)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

These two calls return the start address of the memory used by the operating system to store its internal variables.

7 Events

Events are a mechanism whereby certain interrupts are passed on for easy trapping by the user. The user is thus able to append his or her own routines to react to certain events without the difficulties involved in normal interrupt handling.

7.1 The event vector, EVNTV

Indirection address &220

This vector is called by the operating system during its interrupt routine to provide users with an easy to use interrupt. A number of events may cause the event handling routine to be called via this vector but unlike an interrupt, the reason for the call is passed to the routine. The value in the accumulator indicates the type of event:

Event No.	Cause of Event
0	output buffer becomes empty
1	input buffer becomes full
2	character entering input buffer
3	ADC conversion complete
4	start of VSYNC
5	interval timer crossing zero
6	ESCAPE condition detected
7	RS423 error detected
8	Econet event
9	User event

The event handling routine should not enable interrupts and not last for more than about 2 milliseconds. So that event handling routines may be daisy chained they should preserve registers and return using the old vector contents.

Output buffer empty **0**

This event enters the event handling routine with the buffer number (see chapter 9) in X. It is generated when a buffer becomes empty (i.e. just after the last character is removed).

Input buffer full **1**

This event enters the event handling routine with the buffer number (see chapter 9) in X. It is generated when the operating system fails to enter a character into a buffer because it is full. Y contains the character value which could not be inserted.

Character entering input buffer **2**

This event is normally generated by a key press and the ASCII value of the key is placed in Y. It is generated independently of the input stream selected.

ADC conversion complete **3**

This event is generated when an ADC conversion is completed on a channel. The event handling routine is entered with the channel number on which the conversion was made in Y.

Start of vertical sync **4**

This event is generated 50 times per second coincident with vertical sync. One use of this event is to time the change to a 6845 or video ULA register so that the change to the screen occurs during fly back and not while the screen is being refreshed. This avoids flickering on the screen.

Interval timer crossing zero **5**

This event uses the interval timer (see OSWORD calls &3 and &4, chapter 19). This timer is a 5 byte value incremented 100 times per second. The event is generated when the timer reaches zero.

ESCAPE condition detected **6**

When the ESCAPE key is pressed or an ESCAPE is received from the RS423 (if RS423 ESCAPEs are enabled) this event is generated.

RS423 error event **7**

This event is generated when an RS423 error is detected. It is entered with the 6850 status byte shifted right by one bit in the X register and the character received in Y.

Network error event **8**

This event is generated when a network event is detected. If the net expansion is not present then this could be used for user events.

User event **9**

This event number has been set aside for the user event. This is most usefully generated from a user interrupt handling routine to enable other user software to trap an interrupt easily (e.g. an event generated from an interrupt driven utility in paged ROM). An event may be generated using OSEVEN, see section 7.3.

7.2 Enable/disable event OSBYTE calls

Call address &FFF4
Indirected through &20A
A=&E (14) - enable event
A=&D (13) - disable event

Entry parameters:
X contains the event number, Y=0

On exit:
A is preserved
X contains the old enable state
(0=disabled, >0=enabled)
Y and C are undefined

7.3 OSEVEN, generate event routine

Call address &FFBF
No indirection address

This routine has been implemented on the BBC microcomputer and the Electron but is not available to software running on a second processor. The user event may be generated using this routine.

Entry parameter:

Y=event number

On exit:

C=0, if and only if the event was enabled

7.4 An example using events.

The program listed below uses event number 2, the character entering input buffer event. This event is often referred to as the keyboard event because input is taken from the keyboard by default.

```
10 OSWORD=&FFF1
20 EVNTV=&220
30 DIM MC% 100
40 DIM sound_pars 8
50 FORI=0 TO 3 STEP3
60   P%=MC%
70   [
80         OPT I
90         PHP
100        PHA
110        TXA
120        PHA
130        TYA
140        PHA                          \ save registers
150        STY sound_pars+4             \ SOUND pitch=key ASCII value
160        LDX #sound_pars AND 255
170        LDY #sound_pars DIV 256
180        LDA #7
190        JSR OSWORD                   \ perform SOUND command
200        PLA
210        TAY
220        PLA
230        TAX
240        PLA
250        PLP                          \ restore registers
260        RTS                          \ return from event handler
270   ]
280   NEXTI
290 ?EVNTV=MC% AND &FF
300 EVNTV?1=MC% DIV &100
310 !sound_pars=&FFF50001
320 sound_pars!4=&00010000 :REM    set up SOUND 1,-11,x,1
```

```
330 *FX 14,2
340                              :REM    enable keyboard event
```

When this program has been run, each key press causes a sound to be made in addition to its normal effect. The pitch of this sound is dependent on the key pressed. This event handling routine does not check the identity of the event calling it and so enabling events other than the keyboard event may have some curious consequences.

8 Interrupts

An interrupt on an Acorn BBC series micro is much the same thing as an interrupt to a person. Consider the analogy of a man working at his desk writing a letter (a foreground task). Suddenly the phone rings (an interrupt); the man then stops his writing and answers the phone (servicing the interrupt). Having answered the phone the man then returns to writing his letter where he left off.

In the BBC microcomputer the main objective or foreground task is running the currently selected language such as BASIC or a ROM based application like a word processor. Interrupts are generated 100 times a second by a clock within the system VIA chip. This regular interrupt enables the operating system to perform background tasks such as modifying the current SOUND in accordance to its ENVELOPE definition and incrementing the TIME value. An interrupt is also generated each time a key is pressed so that the operating system can insert the key value into the type-ahead input buffer.

Interrupts are generated by a number of hardware devices which require immediate attention by their servicing software. On the 6502 microprocessor there are two types of interrupts, maskable and non-maskable. The maskable type of interrupt can be prevented from occurring by setting the disable interrupts flag (using a CLI instruction). A non-maskable interrupt (NMI) on the other hand is never ignored and as such is reserved for hardware devices that always require immediate attention like a network or a hard disc. For more information about NMIs see memory allocation chapter 12 and paged ROM service calls section 17.4.1.

An interrupt is physically generated by a signal being sent to one of the pins on the 6502 by a device which requires attention. The 6502 then finishes the execution of its current instruction, saves the status register and the program counter on the stack (ready for an RTI instruction) and JMPs (jumps) to a fixed location using vectors contained in the operating system ROM. When a maskable interrupt is received (on the IRQ pin) and the interrupt disable flag is not set a JMP (&FFFE) is performed. When a non-maskable interrupt is received (on the NMI pin) a JMP (&FFFA) is performed, regardless of the state of the interrupt flag. A maskable interrupt can be generated by software which executes a BRK instruction. The interrupt handling routine can distinguish a BRK interrupt by examining the BRK flag in the status register.

8.1 Non Maskable Interrupts

In the case of an NMI being received, the address at &FFFA points to &D00 which is an area of RAM reserved for the immediate NMI handling routine. This routine may then continue in a paged ROM. The NMI is not used on unexpanded machines but is used by the Econet and Disc filing systems and their hardware.

8.2 Maskable Interrupts

The address contained in locations &FFFE and &FFFF points to code within the operating system which handles the maskable interrupts.

The first thing this code checks is whether this interrupt was caused by a BRK instruction (by testing the BRK flag), if so the interrupt is passed to the BRKV as described in section 8.13.

After this the interrupt routine indirects through IRQ1V (&204). This first vector allows a user interrupt handling routine to intercept the interrupt with a very high priority. The contents of this vector normally point to the main body of the interrupt handling routine in the operating system but before an RTI instruction is executed a final indirection is performed through IRQ2V to allow a low priority interception. When the RTI instruction is executed the foreground task is re-started.

8.3 The operating system interrupt handling routine

When an interrupt occurs the handling routine does not know what has caused it. The routine must first disable interrupts by setting the interrupt flag to prevent another occurring and then poll (that is 'ask') each device which may have caused the interrupt to find out if it needs servicing.

The OS routine attempts to service the interrupt in the following order:

6850 serial chip
(1) RS423 or cassette
System 6522 VIA
(2) V sync
(3) Centi-second timer
(4) ADC conversion
(5) Keyboard
User 6522 VIA
(6) Printer port

8.4 Serial system interrupts

The 6850 asynchronous communications interface adapter chip is used for the cassette and RS423 input and output. At any one time either the cassette system or the RS423 system has control of the chip but never both.

Interrupts are generated by the 6850 in three circumstances:

(a) when a character is received
(b) when a character is transmitted
(c) when the tone at the end of a cassette block is discontinued

To determine whether the 6850 has instigated the current interrupt or not, the 6850 status register (Sheila &08, &FE08) has to be examined. The bits in this register have the following significance.

bit 0	set when a receiver interrupt is generated
bit 1	set when a transmit interrupt is generated
bit 2	set when a DCD (data carrier detect) interrupt is made
bit 3	set if 6850 is not clear to send (CTS)
bit 4	framing error, only valid if bit 0 set
bit 5	receiver over run, only valid if bit 0 set
bit 6	parity error, only valid if bit 0 set
bit 7	set if 6850 generated the current interrupt

The OS examines bit 7 of this register and if set goes on to service the 6850 interrupt. The 6850 interrupt state is cleared by writing to the chip's Transmit Data Register or reading from its Receive Data Register.

Cassette system interrupt use:

Transmitter interrupts are generated when the next byte due to be output is required.

Receiver interrupts indicate that a byte has been read from tape and should be put in memory.

The DCD interrupt marks the end of a block and is used when skipping files (e.g. during a *CAT)

RS423 system interrupt use:

Transmitter interrupts are generated when the 6850 requires another byte of data for output. The source of this byte will be the RS423 transmit buffer or the printer buffer (if RS423 printer selected). If the interrupt handling routine discovers that both buffers are empty it will flag the RS423 system as not busy in response to OSBYTE &BF.

In response to a receiver interrupt the byte read will be placed in the RS423 receive buffer. Bit 7 of the OS copy of the 6850 (Readable using OSBYTE &9C) is effectively a flag enabling RS423 input and a byte read will only be placed in the buffer if this bit is set. Should a receive error be flagged event 7 will be generated and the character ignored (unless OSBYTE &E8 has been used to mask out the error flags). If RS423 input has been suppressed using OSBYTE &CC the character will be ignored. As the buffer becomes full the RTS line is pulled up to prevent the external device from sending any more characters. This occurs when the free-space remaining in the buffer reaches the level defined using OSBYTE &CB.

A DCD interrupt will only occur when RS423 has been directed to the cassette port following the use of OSBYTE &CD. This interrupt is normally cleared by reading form the 6850 receive register. The interrupt servicing routine then generates event number 7 (RS423 receive error).

The interrupt mask for the 6850 status register may be accessed using OSBYTE &E8. The interrupt handling routine ANDs this value with the 6850 status register and any bit cleared by this is ignored by the operating system. The user is then responsible for clearing this interrupt condition when the interrupt is passed to IRQ2V. Clearing the interrupt condition may be achieved either by writing to the transmit register or reading from the receive register.

For detailed information about the 6850 ACIA the reader should refer to the relevant data sheet.

8.5 System VIA interrupts

The system VIA is used to interface a number of internal hardware devices on the BBC microcomputer.

When an interrupt is generated by the system VIA the value of the status register indicates which device caused the interrupt.

bit	interrupt cause if bit set
0	key has been pressed
1	V-Sync has occurred on the video system
2	system VIA shift register times out
3	a light pen strobe has occurred off screen
4	an ADC conversion has been completed
5	timer 2 has timed out (used by speech system)
6	timer 1 has timed out (100Hz interrupt)
7	master interrupt flag (0-6 invalid if not set)

The operating system will ignore any of these interrupts if the appropriate bit in the interrupt mask (see OSBYTE &E9) is set. Masked interrupts are passed onto IRQ2V where a user routine should clear the interrupt condition by writing to the interrupt flag register of the system VIA with the relevant interrupt bit set.

The operating system interrupt handling routine services system VIA interrupts in the following way:

A key pressed interrupt will cause the operating system to mark the key value as the current key. This will be serviced during the next timer interrupt.

A vertical sync interrupt is used to time the duration and change over of two-colour flashing. The change in colour thus occurs during fly back and flickering is avoided. This interrupt is also used to time out the cassette and RS423 systems. This enables the other user of the 6850 to claim it after half a second of inactivity from the existing user. Event number 4 (V-sync) is generated by this interrupt.

The shift register interrupt is not used by the operating system and if generated is passed onto IRQ2V.

The ADC conversion completed interrupt causes the operating system to copy the newly converted value into its work space and the next channel to be converted is initialised. Event number 3 is also generated.

The light pen interrupt is not used by the operating system and is always passed over to IRQ2V.

Timer 2 is used by the speech system to count transitions of the speech ready output. When an interrupt occurs an attempt is made to speak another word.

Timer 1 is used to provide regular interrupts at centi-second intervals. The servicing routine performs the following actions in response to this interrupt:

(a) The interrupt is cleared

(b) One of the two internal clock values is incremented. This clock is then marked as the current clock for use as the TIME value. On the next 100Hz interrupt the other clock value is incremented. The dual clock system is implemented to prevent an interrupt changing the TIME value while it is being read.

(c) The interval timer is incremented and if zero event number 5 is generated.

(d) The INKEY timer is decremented.

(e) One centi-second portion of SOUND is processed

(f) If a key interrupt has occurred recently the new key value is inserted into the input buffer and auto-repeat processing is begun.

(g) The speech chip, the 6850 and the ADC converter are checked to see if any interrupts have been missed.

Disabling interrupts for longer than about 2ms will prevent these background tasks being performed and will have undefined effects.

8.6 User VIA interrupts

Port A of this 6522 is used for the parallel printer port while port B is designated as the user port. All interrupts other than the CA1 interrupt are passed to the user via IRQV2. The parallel printer will cause the CA1 interrupt when it is ready to accept a new character. The interrupt mask for the user VIA may be accessed using OSBYTE &E7.

8.7 Intercepting interrupts

Maskable interrupts may be intercepted on the BBC microcomputer and re-directed to a user specified address. This interception entails the changing of a vector.

There are two points at which interrupts may be intercepted; the two vectors are IRQ1V and IRQ2V.

Interrupt Request Vector 1 (IRQ1V)

Indirects through &204,5

This is the highest priority vector through which all maskable interrupts are indirected. Where at all possible the use of this vector should be avoided and the lower priority, IRQ2V, used instead.

Interrupt Request Vector 2 (IRQ2V)

Indirects through &206,7

This vector is normally called from the operating system interrupt handling routine when unrecognised interrupts occur as described above.

Several points should be considered when producing interrupt handling routines.

(a) When the vector is changed to point at the new user supplied routine, the previous vector contents should be saved. This enables the user routine to pass on any unrecognised interrupts to other routines which may have previously intercepted the vector.

(b) Interrupts should be disabled using a SEI instruction while the interrupt vectors are being changed. This will prevent the operating system interrupt handling routines using the vectors half way through a change.

(c) When the user routine is entered the stack will contain the status register and a return address ready for a return using an RTI instruction. The X and Y registers will be unchanged and the accumulator contents will have been stored in zero page location &FC.

(d) Operating system calls should not be used from within an interrupt handling routine. This is because these routines often use work space memory locations and enable interrupts. If a second interrupt occurs while the first interrupt is being serviced these memory locations will be over written and so not allow the original interrupt servicing to occur correctly.

(e) The users interrupt routine should be re-entrant. This means that if there is a possibility of interrupts being enabled during the routine (e.g. because it is very long), the code can be run again without affecting the first foreground interrupt. This can only be achieved by pushing the contents of the X and Y registers and the old accumulator contents (stored in &FC) onto the stack, and restoring them after the call. It is also important to ensure that no fixed memory locations are used for storing variables, since these will be over-written by an interrupting routine.

8.8 Read/write User 6522 IRQ bit mask OSBYTE call

Call address &FFF4
Indirected through &20A
A=&E7 (231)

This location is reserved for future expansion on the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old mask is returned in X.

This location contains a software copy of the interrupt bit mask for the user VIA chip (6522).

Default value &FF.

8.9 Read/write 6850 IRQ bit mask OSBYTE call

Call address &FFF4
Indirected through &20A
A=&E8 (232)

This location is reserved for future expansion on the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old mask value is returned in X.

This location contains a software copy of the interrupt bit mask for the serial communications chip (6850).

Default value &FF.

8.10 Read/write System 6522 IRQ bit mask OSBYTE call

Call address &FFF4
Indirected through &20A
A=&E9 (233)

This location is reserved for future expansion on the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old mask value is returned in X.

This location contains the interrupt bit mask used for the system VIA chip (6522).

Default value &FF.

8.11 Read/write Electron ULA IRQ mask OSBYTE call

Call address &FFF4
Indirected through &20A
A=&CB (203)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old mask value is returned in X.

This location contains the interrupt bit mask used for the Electron ULA.

Default value &0C.

8.12 BRK/error associated calls

The BRK instruction is normally used on Acorn machines to represent an error condition and the BRK vector routine is an error handling routine. In BASIC this error handling routine starts off by putting its house in order and then prints out an error message.

In addition to the use of BRKs for the generation of errors it is often useful in machine code programming to include BRKs (break-points) as a debugging aid.

8.13 The BRK vector &202

When a BRK instruction (op code value 0) is executed a maskable interrupt is generated. The operating system stores the address of the byte following the BRK instruction in &FD and &FE, offers the BRK to paged ROMs with service call &06, stores the ROM number of the currently active paged ROM for recovery using OSBYTE &BA (ROM active at last BRK), restores registers, selects the current language ROM, and then passes the call to the BRKV code.

On Acorn BBC range machines the BRK vector is entered with the following conditions if a BRK instruction is executed:

(a) The A, X and Y registers are unchanged from when the BRK instruction was executed.

(b) A RTI instruction will return execution to the address two bytes after the BRK instruction (i.e. jumps over the byte following the BRK). The RTI instruction also restores the status register value from the stack.

(c) The address of the byte following the BRK instruction is stored in zero page locations &FD and &FE. This address can then be used for indexed addressing .

Error handling BRK routines should not return to the code which executed the BRK but should reset the stack (using a TXS instruction) and JMP into a suitable reset entry point. In fact the convention used by Acorn is to follow the BRK instruction by:

a single byte error number
an error message
a zero byte to terminate the message

and the BRK routine prints out the error name. The BRK handling routine should normally be implemented by the current language. Service paged ROMs should copy a BRK instruction followed by the error number and message down into RAM when wishing to generate an error. This has to be done because otherwise the current language ROM is paged in and the BRK handling routine tries to print out the error message from the wrong ROM. The bottom of page 1 is often used and is quite safe as long as the BRK handling routine resets the stack pointer.

The use of BRKs as break-points in machine code programming can be of great use to the machine code programmer. The example below shows how a BRK handling routine may be used to print out the register values. This routine could be further enhanced by printing out the value of the byte following the BRK instruction which would then give the programmer 256 individually identifiable break-points.

```
10 REM Primitive BRK handling routine
20 DIM code% &100
30 OSASCI=&FFE3
40 OSRDCH=&FFE0
50 BRKV=&202
60 FOR opt%=0 TO 3 STEP 3
70 P%=code%
80 [
90 OPT opt%

100 .init   LDX #brkrt AND &FF  \ load registers with address
110         LDY #brkrt DIV &100
120         SEI                 \ disable interrupts
130         STX BRKV            \ set up BRK vector
140         STY BRKV+1
150         CLI                 \ enable interrupts and return
160         RTS

170 .brkrt  PHA                 \ save A (X and Y not used)
180         STA byte            \ store A in workspace
```

```
190           LDA #ASC"A"             \ register id
200           JSR prntrg              \ print register value
210           STX byte                \ store X in workspace
220           LDA #ASC"X"             \ register id
230           JSR prntrg              \ print register value
240           STY byte                \ store Y in workspace
250           LDA #ASC"Y"             \ register id
260           JSR prntrg              \ print register value
270           JSR new_ln              \ print carriage return
280           JSR OSRDCH              \ wait for key press
290           PLA                     \ restore A
300           RTI                     \ return

310 .prntrg   JSR OSASCI              \ print register id
320           LDA #ASC":"
330           JSR OSASCI              \ print colon
340           JSR space               \ print space
350           LDA #ASC"&"
360           JSR OSASCI              \ print ampersand
370           LDA byte                \ get register value
380           JSR prntbt              \ print hex number
390           JSR space
400           JSR space               \ print two spaces
410           RTS

420 .space    LDA #&20
430           JMP OSASCI              \ print space

440 .new_ln   LDA #&D
450           JMP OSASCI              \ print carriage return

460 .prntbt   PHA                     \ for comments refer to
470           LSR A                   \     previous example
480           LSR A
490           LSR A
500           LSR A
510           JSR nibble
520           PLA
530 .nibble   AND #&0F
540           CMP #&0A
550           BCC number
560           ADC #&06
570 .number   ADC #&30
580           JMP OSASCI

590 .byte     EQUB 0                  \ workspace byte

600 .test     BRK                     \ cause an error
610           EQUB 0                  \ RTI returns to next byte
620           DEX                     \ loop X times
630           BNE test                \ if X=0 loop again
640           RTS

650 ]
660 NEXT
670 CALL init
680 A%=1:X%=8:Y%=&FF:CALL test
```

8.14 Read ROM no. active at last BRK OSBYTE call

Call address &FFF4
Indirected through &20A
A=&BA (186)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old ROM number value is returned in X.

The operating system records the number of the paged ROM active when a BRK occurs before selecting the current language ROM for BRK handling.

9 Buffers control and management

The operating system uses buffers for keyboard input, RS423 input and output, the printer, the sound system (4 buffers) and the speech system. These buffers contain data which should be processed by the various routines. Even though the servicing routine may not be able to respond to the request immediately the calling routine returns (unless the buffer is full) and is able to get on with its foreground task. While a buffer contains a queue of data for processing, the interrupt routine (the background task) ensures that the relevant routines service this data.

In this way the user is able to type ahead when the machine is unable to respond immediately, and may initiate sounds which then continue while he issues further commands.

Buffers operate on a first in first out (FIFO) basis for obvious reasons.

The Acorn BBC range of machines use the following numbers as buffer id's:

title	buffer no.
keyboard buffer	0
RS423 input buffer	1
RS423 output buffer	2
printer buffer	3
SOUND channel 0 buffer	4
SOUND channel 1 buffer	5
SOUND channel 2 buffer	6
SOUND channel 3 buffer	7
speech buffer	8

On the BBC microcomputer or the Electron memory is reserved for each of these buffers even though the software/hardware using the buffer may not be present. The buffer maintenance calls still service these buffers but the contents will not be processed by the relevant service routine. The expansion software/hardware will use the appropriate buffer when installed. Thus when the speech expansion is fitted on a BBC microcomputer the speech buffer is used and on an Electron with a Plus 1 the printer buffer is used.

9.1 Insert value into buffer vector, INSV

Indirection address &22A

This vector contains the address of a routine which inserts a value into a selected buffer.

Entry parameters:
A=value to be inserted
X=buffer id

On exit:
A and X are preserved
Y is undefined

C flag is set if insertion failed (i.e.buffer full)

9.2 Remove value from buffer vector, REMV

Indirection address &22C

This vector contains the address of a routine which removes a value from the selected buffer. This routine may also be used to examine the next character to be removed from a buffer without actually removing it.

Entry parameters:
X=buffer id
V=1 (overflow flag set) if only examination requested

On exit:
A contains next byte to be removed (examination call)
(A undefined for removal call)
X is preserved
Y contains the value of the byte removed from the buffer
(Y undefined for examination call)
C flag is set if buffer when call made

9.3 Count/purge buffer vector, CNPV

Indirection address &22E

This vector contains the address of a routine which may be used to clear the contents of a buffer or to return information about the free space or contents of a buffer.

Entry parameters:
X=buffer id
V=1 (overflow flag set) to purge buffer
V=0 (overflow flag clear) for count operation
C=1 count operation returns amount of free space
C=0 count operation returns length of buffer contents

On exit:

X and Y contain value of count (low byte, high byte)
X and Y are preserved for a purge operation
A is undefined
V and C are preserved

9.4 Using the buffer vectors

It should be noted that none of the buffer maintenance routines check for valid buffer id's. Using a buffer id outside the assigned range will have undefined effects unless specifically intercepted.

None of these vectors are implemented on second processors and so none of the buffer maintenance calls are sent across the Tube. Calls using the buffer vectors should always be made by code resident in the i/o processor. It should be noted that considerable manipulation of the buffers may be carried out using OS routines such as OSBYTE, OSWRCH, OSWORD etc. which may affect buffer contents either directly or indirectly. Routines intercepting these vectors must always be resident on the i/o processor, ideally in service type paged ROMs.

The program below illustrates how the buffer vectors can be intercepted to implement a much larger printer buffer. The standard printer buffer is only &100 bytes long and since printers tend to be quite sluggish peripherals this buffer rapidly fills up. A buffer is required which will hold a reasonable sized listing, or a document before filling up and refusing to accept further input. Having placed the item for printing in an enlarged buffer, the user may return to word processing or programming leaving the operating system to get on with the printing.

The routine used below creates a buffer of variable size as defined by the variable '`size`'. The utility value of this program is limited. For the reasons given above it will only work when run on a non-Tube machine. It will only work as long as its code is not corrupted; this means that renumbering the program after it has been run will crash the machine since BASIC tramples all over area originally reserved for the assembled code. Similarly another language ROM is unlikely to allow the routine to run in peace. If this routine becomes corrupted the machine will become totally disabled because each time a key is pressed this routine is called. Experimenting with this example will provide valuable experience in the use of critical operating system routines. One note of warning however, be sure to save a copy of the program before trying to run it; it is quite possible for the program to corrupt itself or

even crash the machine irrevocably so that a power on reset is required (that is, the machine will have to be turned off, and then on again).

This program consists of three main routines which intercept the buffer maintenance calls for the printer buffer. Calls for any of the other buffers are carefully handed on to the original routines pointed to by the contents of the buffer vectors. An area of RAM is reserved for use as a buffer by using a DIM statement. Four bytes of zero page memory are used to house two 16 bit pointers. One pointer is used as an index for the insertion of values into the buffer, and the other pointer is used as an index for the removal of bytes. When a pointer reaches the end of the buffer it is pointed to the beginning again. In this way the two pointers cycle through the buffer space. A full buffer is detected by incrementing the input pointer and comparing it to the output pointer. If the two pointers are equal then the buffer is full, the character cannot be inserted and the input pointer is restored. If after the removal of a character the output pointer becomes equal to the input pointer then the buffer is now empty. By using this system the full size of the buffer is always available to contain data.

```
10 REM user printer buffer routine

20 MODE7
 30 size=&2000
 40 DIM buffer size
 50 DIM code% &400
 60 INSV=&22A
 70 RMV=&22C
 80 CNPV=&22E
 90 ptrblk=&80:!ptrblk=buffer+buffer*&10000
100 ip_ptr=ptrblk:op_ptr=ptrblk+2

110 FOR I=0 TO 3 STEP 3
120 P%=code%
130 [
140 OPT I

150 .init     LDA INSV              \ make copies of old vector
160           STA ret1              \  contents to pass on calls
170           LDA INSV+1
180           STA ret1+1
190           LDA RMV
200           STA ret2
210           LDA RMV+1
220           STA ret2+1
230           LDA CNPV
240           STA ret3
250           LDA CNPV+1
260           STA ret3+1

270           LDX #ins AND &FF      \ store address of new
280           LDY #ins DIV &100     \  routines in vectors
290           SEI                   \ disable interrupts
```

```
300           STX INSV
310           STY INSV+1
320           LDX #rem AND &FF
330           LDY #rem DIV &100
340           STX RMV
350           STY RMV+1
360           LDX #cnp AND &FF
370           LDY #cnp DIV &100
380           STX CNPV
390           STY CNPV+1
400           CLI                       \ enable interrupts
410           RTS                       \ finished

420 .wrkbt    EQUB 0                    \ byte of RAM workspace

430 .ret1     EQUW 0                    \ reserve space for vectors
440 .ret2     EQUW 0
450 .ret3     EQUW 0

460 .wrngbf1 PLP:PLA:JMP (ret1)   \restore S & A, call OS

470 \ New insert char. into buffer routine

480 .ins      PHA:PHP                   \ save A and status register
490           CPX #3                    \ is buffer id 3 ?
500           BNE wrngbf1               \ if not pass to old routine
510           PLP                       \ not passing on, tidy stack
520           LDA ip_ptr                \ A=lo byte of input pointer
530           PHA                       \ store on stack
540           LDA ip_ptr+1              \ A=hi byte of input pointer
550           PHA                       \ store on stack
560           LDY #0                    \ Y=0 so ip_ptr incremented
570           JSR inc_ptr               \  by the inc_ptr routine
580           JSR compare               \ compare the two pointers
590           BEQ insfail               \ if ptrs equal, buffer full
600           PLA:PLA:PLA               \ don't need ip_ptr copy now
610           STA (ip_ptr),Y            \ A off stack, insrt in bufr
620           CLC                       \ insertion success, C=0
630           RTS                       \ finished

640 .insfail PLA                        \ buffer was full so must
650           STA ip_ptr+1              \  restore ip_ptr which was
660           PLA                       \  stored on the stack
670           STA ip_ptr
680           PLA
690           SEC                       \ insertion fails so C=1
700           RTS                       \ finished

710 .wrngbf2 PLP:JMP (ret2)             \ restore S, call OS

720 \ New remove char. from buffer routine

730 .rem      PHP                       \ save status register
740           CPX #3                    \ is buffer id 3 ?
750           BNE wrngbf2               \ if not use OS routine
760           PLP                       \ restore status register
770           BVS examine               \ V=1, examine not remove

780 .remsr    JSR compare               \ compare i/p and o/p ptrs
790           BEQ empty                 \ if the same, buffer empty
800           LDY #2                    \ Y=2 so that increment ptr
810           JSR inc_ptr               \  routine inc's op_ptr
```

```
 820           LDY #0                  \ Y=0, for next instruction
 830           LDA (op_ptr),Y          \ fetch character from bufr
 840           TAY                     \ return it in Y
 850           CLC                     \ buffer not empty, C=0
 860           RTS                     \ return

 870 .empty    SEC                     \ buffer empty, C=1
 880           RTS                     \ return

 890 .examine  LDA op_ptr              \ examine only, so store a
 900           PHA                     \  copy of the o/p pointer
 910           LDA op_ptr+1            \  on the stack to restore
 920           PHA                     \  ptr after fetch
 930           JSR remsr               \ fetch byte from buffer
 940           PLA                     \ restcre ptr from stack
 950           STA op_ptr+1            \ (if buffer was empty
 960           PLA                     \  C=1 from fetch call)
 970           STA op_ptr
 980           TYA                     \ examine requires ch. in A
 990           RTS                     \ finished

1000 .wrngbf3 PLP:JMP (ret3)          \ restore S, call OS

1010 \ New count/purge buffer routine

1020 .cnp     PHP                     \ save status reg. on stack
1030          CPX #3                  \ is buffer id 3 ?
1040          BNE wrngbf3             \ if not pass to old subr
1050          PLP                     \ restore status register
1060          PHP                     \ save again
1070          BVS purge               \ if V=1, purge required
1080          BCC len                 \ if C=0, amount in buffer

1090          LDA ip_ptr              \  o/w free space request
1100          PHA
1110          LDA ip_ptr+1            \ store ip_ptr on stack
1120          PHA
1130          LDX #0                  \ X=0 for use as counter
1140          STX wrkbt               \ wrkbt=0 for hi counter
1150          LDY #0                  \ Y=0, so ip_ptr incr'd
1160 .loop1   JSR inc_ptr             \ increment ip_ptr
1170          JSR compare             \ does it equal op_ptr
1180          BEQ finshd1             \ if so count=free space
1190          INX                     \ X=X+1
1200          BNE no_inc              \ if X=0 don't inc wrkbt
1210          INC wrkbt               \ hi byte of count inc'd
1220 .no_inc  JMP loop1               \ loop round again

1230 .finshd1 PLA                     \ restore ip_ptr off stack
1240          STA ip_ptr+1
1250          PLA
1260          STA ip_ptr
1270          LDY wrkbt               \ Y=hi byte of free space
1280          PLP                     \ restore status register
1290          RTS                     \ finished

1300 .len     LDA op_ptr              \ store op_ptr on stack
1310          PHA
1320          LDA op_ptr+1
1330          PHA
1340          LDX #0                  \ X=0 for use as counter
```

```
1350            STX wrkbt                \ wrkbt=0 hi byte of count
1360            LDY #2                   \ Y=2 so op_ptr incremented
1370 .loop2     JSR compare              \ are ptrs equal ?
1380            BEQ finshd2              \ if so buffer empty
1390            JSR inc_ptr              \ increment op_ptr
1400            INX                      \ increment count
1410            BNE no_inc2              \ if X=0 then increment hi
1420            INC wrkbt                \  byte of count
1430 .no_inc2   JMP loop2                \ loop round again

1440 .finshd2   PLA                      \ restore op_ptr off stack
1450            STA op_ptr+1
1460            PLA
1470            STA op_ptr
1480            LDY wrkbt                \ Y=hi byte of length
1490            PLP                      \ restore status register
1500            RTS                      \ finished

1510 .purge     LDA #buffer AND &FF      \ to purge buffer reset
1520            STA ip_ptr               \  o/p and i/p ptrs to
1530            STA op_ptr               \  start of buffer
1540            LDA #buffer DIV &100
1550            STA ip_ptr+1
1560            STA op_ptr+1
1570            PLP                      \ restore status register
1580            RTS                      \ return

1590 \ Increment pointer routine. Y=0 op_ptr, Y=2 ip_ptr

1600 .inc_ptr   CLC                      \ C=0
1610            LDA ptrblk,Y             \ A=?(ptrblk+Y)
1620            ADC #1                   \ A=A+1+C
1630            STA ptrblk,Y             \ ?(ptrblk+Y)=A
1640            LDA ptrblk+1,Y           \ A=?(ptrblk+1+Y)
1650            ADC #0                   \ A=A+0+C
1660            STA ptrblk+1,Y           \ ?(ptrblk+1+Y)=A
1670            CMP #(buffer+size) DIV &100 \ hi byte end of bufr
1680            BNE home                 \ not end of buffer
1690            LDA ptrblk,Y             \ A=low byte of pointer
1700            CMP #(buffer+size) AND &FF \ end of buffer ?
1710            BNE home
1720            LDA #buffer AND &FF      \ if the end of buffer has
1730            STA ptrblk,Y             \ been reached set pointer
1740            LDA #buffer DIV &100     \ to the beginning again
1750            STA ptrblk+1,Y
1760 .home      RTS                      \ return

1770 \ Compare pointers. if equal Z=1 don't care otherwise

1780 .compare   LDA ip_ptr+1
1790            CMP op_ptr+1             \ compare ptr high bytes
1800            BNE return               \ if not equal return
1810            LDA ip_ptr
1820            CMP op_ptr               \ compare pointr low bytes
1830 .return    RTS                      \ return

1840 ]
1850 NEXT
1860 CALL init
```

9.5 Flush specific buffer OSBYTE call

Call address &FFF4
Indirected through &20A
A=&15 (21)

While the unexpanded Electron only has a single sound channel the operating system has been designed to enable the implementation of an external sound system. Each time any of the sound buffers are flushed a paged ROM service call is issued with A=&17. In the unexpanded Electron there is a single effective buffer which may be addressed as any of the four channels. Thus flushing any of the four buffers will extinguish any noise being produced at that time.

Entry parameters:
X=number of buffer to be cleared

On exit:
A and X are preserved
Y and C are undefined

9.6 Flush selected buffer class OSBYTE call

Call address &FFF4
Indirected through &20A
A=&0F (15)

Entry parameters:
X value selects class of buffer

X=0 All buffers flushed
X<>0 Input buffer flushed only

On exit,
Buffer contents are discarded
A is preserved
X, Y and C are undefined

9.7 Read buffer status OSBYTE call

Call address &FFF4
Indirected through &20A
A=&80 (128)

This OSBYTE call also has ADC functions (see chapter 7).

Information about those buffers not used on the unexpanded Electron will be meaningless; these buffers have been implemented for expansion capability.

Entry parameters:
X determines action and buffer number

X=255	(&FF)	keyboard buffer
X=254	(&FE)	RS423 input buffer
X=253	(&FD)	RS423 output buffer
X=252	(&FC)	printer buffer
X=251	(&FB)	sound channel 0
X=250	(&FA)	sound channel 1
X=249	(&F9)	sound channel 2
X=248	(&F8)	sound channel 3
X=249	(&F7)	speech buffer

For input buffers X is returned containing the number of characters in the buffer and for output buffers the number of spaces remaining.

On exit:
A is preserved
C is undefined

9.8 Insert value into buffer OSBYTE call

Call address &FFF4
Indirected through &20A
A=&8A (138)

Entry parameters:
X contains buffer number
Y contains the value to be inserted into buffer

On exit:
C=0 if value successfully inserted
C=1 if value not inserted e.g. if buffer full

A is preserved

9.9 Get character from buffer OSBYTE call

Call address &FFF4
Indirected through &20A
A=&91 (145)

Entry parameters:
X contains buffer number

On exit:
Y contains the extracted character.
If the buffer was empty then C=1 otherwise C=0.

A is preserved

9.10 Examine buffer status OSBYTE call

Call address &FFF4
Indirected through &20A
A=&98 (152)

This call has been implemented differently on old versions of the BBC OS up to and including OS 1.20.

(a) BBC model B microcomputers

Entry parameters:
X contains buffer number (See OSBYTE &15 for numbers)

On exit:

If the buffer is not empty
Y=pointer to next character to be read from the buffer indexed from zero page locations &FA and &FB.
C=0

If the buffer is empty
Y is preserved
C=1

After using this call to examine the next character to be read from a non-empty buffer the instructions "LDA (&FA),Y" will be required. Interrupts should be disabled while the OSBYTE call is made and the buffer examined to prevent any interrupt changing the buffer.

A and X are preserved

(b) Master, B+ and Electron

Entry parameters as above

On exit:

Y=character value read from buffer if buffer not empty
Y is preserved if buffer empty
C=1 if buffer empty otherwise C=0

A and X are preserved

9.11 Insert char. into i/p buffer OSBYTE call

Call address &FFF4
Indirected through &20A
A=&99 (153)

Entry parameters:

X contains buffer number (0 or 1 only)
Y contains the character value

X=0 keyboard buffer
X=1 RS423 input

If RS423 input is enabled and X=1 then RS423 ESCAPEs are suppressed (this is the default state plus OSBYTE call with A=&B5 and X=1/*FX181,1). This is identical to OSBYTE call with A=&8A (*FX 138).

Otherwise if the character to be inserted is not the ESCAPE character (set by OSBYTE &DC/*FX 220) or if ESCAPE characters are to be treated as normal characters (following OSBYTE with A=&E5/*FX 229), then an input event (even if input is from RS423) is caused and the character is inserted into the buffer.

If the character is an ESCAPE character and ESCAPEs are not protected (using OSBYTE &C8/*FX 200) then an ESCAPE event is generated instead of the keyboard event.

On exit:

A is preserved
X, Y and C are undefined

10 Escape related calls

The following calls have effects on the functioning of the ESCAPE key or the escape condition.

10.1 Clear escape condition OSBYTE call

Call address &FFF4
Indirected through &20A
A=&7C (124)

This call clears any ESCAPE condition without any further action. The Tube is informed if active. If there was not an ESCAPE condition to clear the X register is returned with value zero.

On exit:

A and Y are preserved
X=0 if there was no ESCAPE condition to clear
X=&FF if an ESCAPE condition was cleared
C is undefined

10.2 Set escape condition OSBYTE call

Call address &FFF4
Indirected through &20A
A=&7D (125)

This call partially simulates the ESCAPE key being pressed. The Tube is informed (if active). An ESCAPE event is not generated.

On exit:

A, X and Y are preserved
C is undefined

10.3 Clear escape + effects OSBYTE call

Call address &FFF4
Indirected through &20A
A=&7E (126)

This call attempts to clear the ESCAPE condition. All active buffers will be flushed, any open EXEC files closed, the VDU paging counter will be reset and the VDU queue will be reset.

On exit:

X=&FF if the ESCAPE condition cleared
X=0 if no ESCAPE condition found

A is preserved
Y and C are undefined

10.4 Read/write escape disable OSBYTE call

Call address &FFF4
Indirected through &20A
A=&C8 (200)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

bit 0=0	Normal ESCAPE action
bit 0=1	ESCAPE disabled unless caused by OSBYTE &7D/125
bits 1 to 7=0	Normal BREAK action
bits 1 to 7=1	Memory cleared on BREAK

10.5 Read/write ESCAPE character OSBYTE call

Call address &FFF4
Indirected through &20A
A=&DC (220)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old character value is returned in X.

This location contains the ASCII character (and key) which will generate an ESCAPE condition or event.

e.g. *FX 220,65 will make A the ESCAPE key.

Default value &1B (27).

10.6 Read/write ESCAPE key status OSBYTE call

Call address &FFF4
Indirected through &20A
A=&E5 (229)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old status is returned in X.

If this location contains 0 then the ESCAPE key has its normal action. Otherwise the currently selected ESCAPE key is treated as an ASCII code.

10.7 Read/write ESCAPE effects OSBYTE call

Call address &FFF4
Indirected through &20A
A=&E6 (230)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

If this location contains 0 then when an ESCAPE is acknowledged (using OSBYTE &7E/*FX 126) then :-

EXEC file is closed (if open)
Purge all buffers (including input buffer)
Reset paging counter (lines since last halt)
Reset VDU queue
Any current soft key expansion is cleared

If this location contains any value other than 0 then ESCAPE causes none of these.

11 An Introduction to Hardware

Most users of the BBC microcomputer will be familiar with BASIC programs, but from BASIC the hardware is virtually invisible. Commands are provided to deal with output to the screen, input from the keyboard and analogue to digital converter, plus all of the other hardware. The same applies to machine code to a large extent through the use of operating system commands. However, a much more detailed understanding of the hardware and how it can be controlled directly from machine code programs is very useful and allows certain features to be implemented which would have been impossible in BASIC.

The remainder of this book satisfies the requirements of both those who wish to use the hardware features already present on the computer, and those who wish to add their own hardware. Each major chapter covers a particular aspect of hardware and it's associated software support, and should be delved into as required. All of the standard hardware features available on the BBC microcomputer are outlined in detail. Wherever possible, it is better to use operating system routines for controlling the hardware as this will ensure compatibility with the entire Acorn-BBC range of computers and peripherals. For those who wish to add their own hardware, full details on using the USER port and 1MHz BUS are supplied.

Figure 11.1 illustrates the major blocks of the computer's circuitry. This has essentially remained constant for the model B, B Plus and Master series, the major differences being in the particular chips used to implement each block on different machines.

At the centre of the system is the central processing unit (CPU) or microprocessor. This is the chip which executes all the programs including BASIC. It is connected to the rest of the system via three buses. These are the data bus, the address bus and the control bus. For clarity on the diagram, these buses are all compressed into one which is represented by the double lines terminated with arrows at each major block. Originally the model B used a 6502A CPU, but the newer Master takes advantage of chip developments and uses the 65C12 which supports all 6502 instructions AND extra new instructions. These extra instructions are covered in chapter 6. Throughout the rest of this book we will use '6502' to refer to the generic CPU family of which both the 6502A and 65C12 are a part.

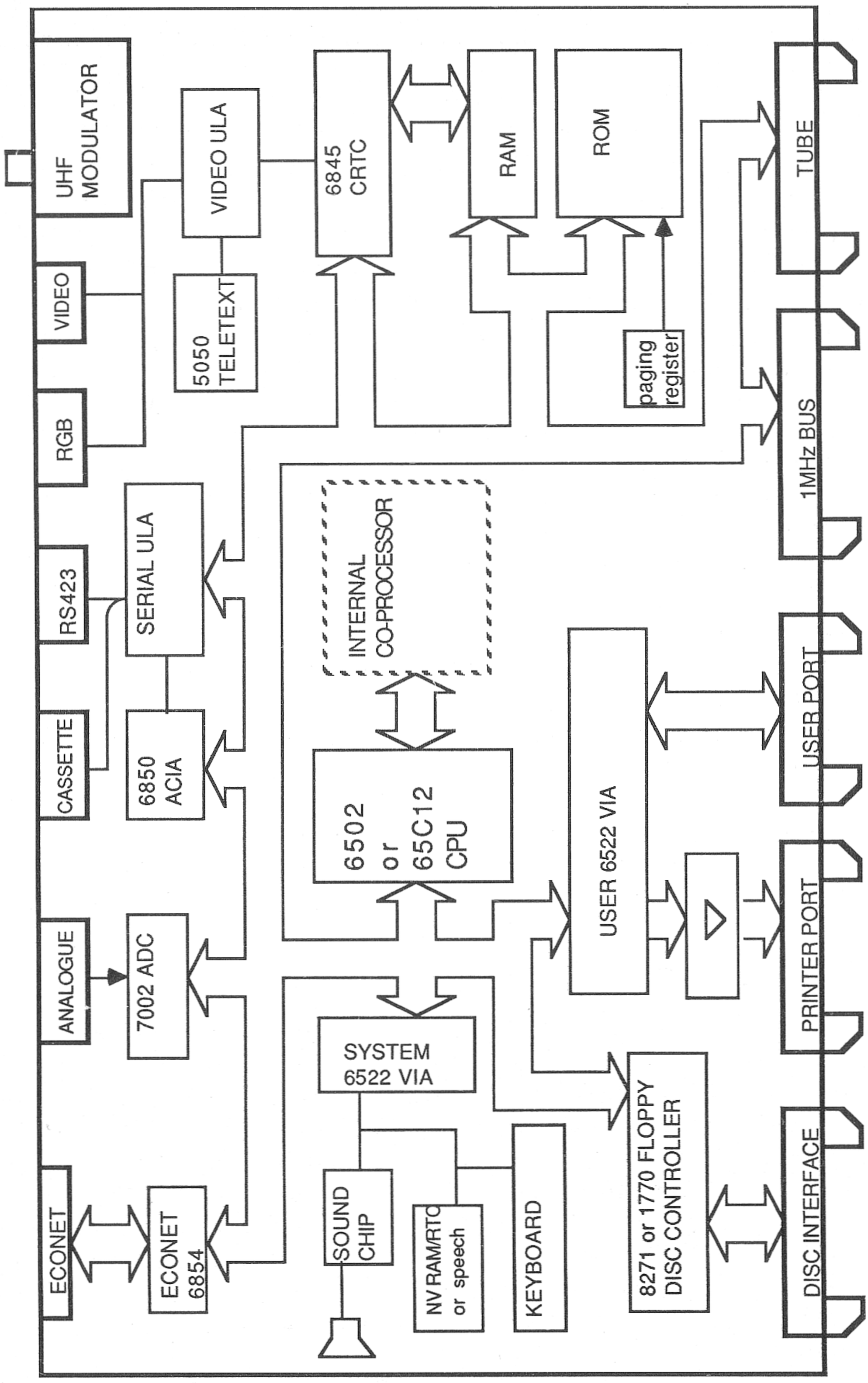


FIG 11.1 - The System Block Diagram

A *bus* is simply a number of electrical links connected in parallel to several devices. The CPU spends its time reading and writing data to

these devices. The communication protocols which enable this transfer of data to take place are set up by the control, address and data buses. In the case of the address bus, there are 16 separate lines which allow 2^16 (65536) different combinations of 1's and 0's. The maximum amount of directly addressable memory on a 6502 is therefore 65536 bytes. The data bus consists of 8 lines, one for each bit of a byte. Any number between 0 and &FF (255) can be transferred across the data bus. Communication between the peripherals, memory and the CPU occurs over the data bus. The CPU can either send out a byte or receive a byte. The data bus is therefore called a *bidirectional* bus because data flows in any one of two directions. The address bus is unidirectional since the 6502 provides, but cannot receive addresses. Note that some address buffers are included in the video circuit to allow either the 6845, 5050 or 6502 to provide addresses for system random access memory (RAM).

In order to control the direction of data flow on the data bus, a read or write signal is provided by the control bus. Hardware connected to the system can thereby determine whether it is being sent data or is meant to send data back to the CPU. The other major control bus functions are those of providing a clock and generating interrupts and resets. The clock signal keeps all the chips running together at the same rate. The RESET line allows all hardware to be initialised to some predefined state after a reset. An interrupt is a signal sent from a peripheral to the 6502 which requests the 6502 to look at that peripheral. Two forms of interrupt are provided. One of these is the interrupt request (IRQ) which the 6502 can ignore under software control. The other in the non-maskable interrupt (NMI) which can never be ignored. Only very important devices which require immediate attention (like the disc controller) can use NMIs.

When power is first applied to the system, a reset is generated to ensure that all devices start up in their reset states. The 6502 then starts to get instructions from the MOS ROM. These instructions tell the 6502 what it should do next. A variety of different instructions exist on the 6502. The basic functions available are reading or writing data to memory or an input/output device and performing arithmetic and logical operations on the data. Once the MOS (machine operating system) program is entered, this piece of software gains full control of the system.

SHEILA and the system hardware

In 6502 systems the hardware is *memory mapped* which means that any hardware device registers appear in the main memory address space. Page &FE (the 256 bytes of memory starting at &FE00) is reserved especially for the system hardware inside the BBC microcomputer. The

special name of "SHEILA" has been assigned to this page of memory. Two other special pages are &FC (called "FRED") and &FD (called "JIM"). FRED and JIM are concerned with external user hardware attached to the one megahertz bus and the cartridge socket on the Master. They are dealt with in chapter 23.

The remainder of the 64K memory map is fully occupied by a combination of RAM and ROM. To overcome the limitations which would otherwise be imposed by such a small memory address space, this memory is paged and overlaid in various ways to achieve a larger effective addressing capability (see chapter 12).

The model B could address the 16K MOS ROM, 4 x 16K paged ROMs and 32K RAM. The B Plus received the additional capacity for installing 11 x 16K logical paged ROMs on the main board together with 20K shadow video RAM and 12K sideways RAM. The Master improved on this by adding another 64K RAM typically organised as 4 x 16K paged RAMs and an external cartridge socket allowing further expansion of RAM and/or ROM.

All the machines have the facility for adding an external co-processor (commonly referred to as a second processor). The Master additionally has internal connectors which allow for another co-processor to be mounted internally. These co-processors can be based around any microprocessor, and have their own associated memory, but none of their own I/O devices. All communication between co-processors and the outside world passes through the 'Tube' interface to the BBC Micro. In this manner, the extensive I/O capabilities of the BBC Micro can be accessed.

In the following chapters, all of the devices attached to Sheila are described in detail. The table below shows the memory map of Sheila and notes the major differences between the various Acorn-BBC range of machines:

SHEILA offset	Integrated Circuit	Description	model B	B+	Master	compact	Electron
&00 - &0F	Electron ULA	various hardware functions					•
&00 - &07	6845 CRTC	Video controller	•	•	•	•	
&08 - &0F	6850 ACIA	Serial controller	•	•	•	•*	
&10 - &1F	Serial ULA	Serial system chip	•	•	•	•	
(&18 - &1A)	µPD7002	Analogue to digital converter			•		
&20 - &21	Video ULA	Video system chip	•	•	•	•	
&24	Floppy disc control reg.				•	•	
&28	WD1770	Floppy disc control chip			•	•	
&30	74LS161 (ROMSEL)	Paged ROM select register	•	•	•	•	
&34	ACCCON	Memory access control register		•	•	•	
&38	INTOFF	Network NMI disable			•	•	
&3C	INTON	Network NMI enable			•	•	
&40 - &5F	6522 VIA	System VIA	•	•	•	•	
		controls keyboard, sound	•	•	•	•	
		optional speech processor	•	•			
		CMOS battery backed clock & RAM			•		
		128 or 256 byte EEPROM				•	
&60 - &7F	6522 VIA	User VIA (user port)	•	•	•	•	
&80 - &9F	8271 FDC	Floppy Disc controller	•	•			
	1770 FDC	Floppy Disc controller		•			
		[Third party h/w e.g. Modem]			•		
&A0 - &BF	68B54 ADLC	Econet controller	•	•	•	•	
&C0 - &DF	µPD7002	Analogue to digital converter	•	•			
	Network interface	Econet interface			•	•	
&E0 - &FF	Tube ULA	Tube system interface	•	•	•	•	

(* present only if serial expansion fitted)

12 Memory

This chapter is primarily concerned with the physical access to memory. The memory hardware and operating system calls controlling memory access are described here. The memory allocated for use by specific sub-systems is described in the relevant chapters. For information about how memory is allocated and used by the operating system refer to section 6.6.

12.1 Memory map overview

The 6502 series of microprocessor found at the heart of all Acorn-BBC computers can address up to 64K bytes of memory and memory-mapped hardware. Acorn have carefully allocated sections of this limited sized memory map to provide a defined environment within which all programs can operate.

As can be seen from the memory map illustrated in fig. 12.1, all Acorn-BBC machines share a similar basic arrangement for the memory, with additional features on the B Plus and Master machines. The shared features provide for the lower 32K of memory to be RAM and the upper 16K to be mainly the MOS ROM with three 256 byte pages allocated to memory-mapped hardware (see hardware introduction). The 16K section in between contains paged ROM and RAM.

On the model B the first 12K of RAM is used for variable storage, system use and program storage. The remaining 20K of RAM is allocated between the screen and program storage depending upon the screen mode used. Mode 0 uses the whole of this 20K whilst mode 7 uses only 1K. To overcome this severe limitation on available RAM, both the B Plus and Master computers have a 20K 'shadow' screen which can operate without occupying the RAM in the main memory map.

To achieve more space than 16K for application programs in ROM, the ROMs are paged. A 4 bit paging register allows up to 16 different ROMs to be addressed, considerably increasing the amount of software which can reside in the machine. The Master is even more sophisticated, being able to provide 4K of MOS workspace and 8K of filing system RAM outside of the lower 32K RAM. The additional filing system RAM overcomes the problem which occurred on earlier machines whereby some filing systems grabbed parts of the lower 12K of RAM for their own use, reducing the RAM available to the user.

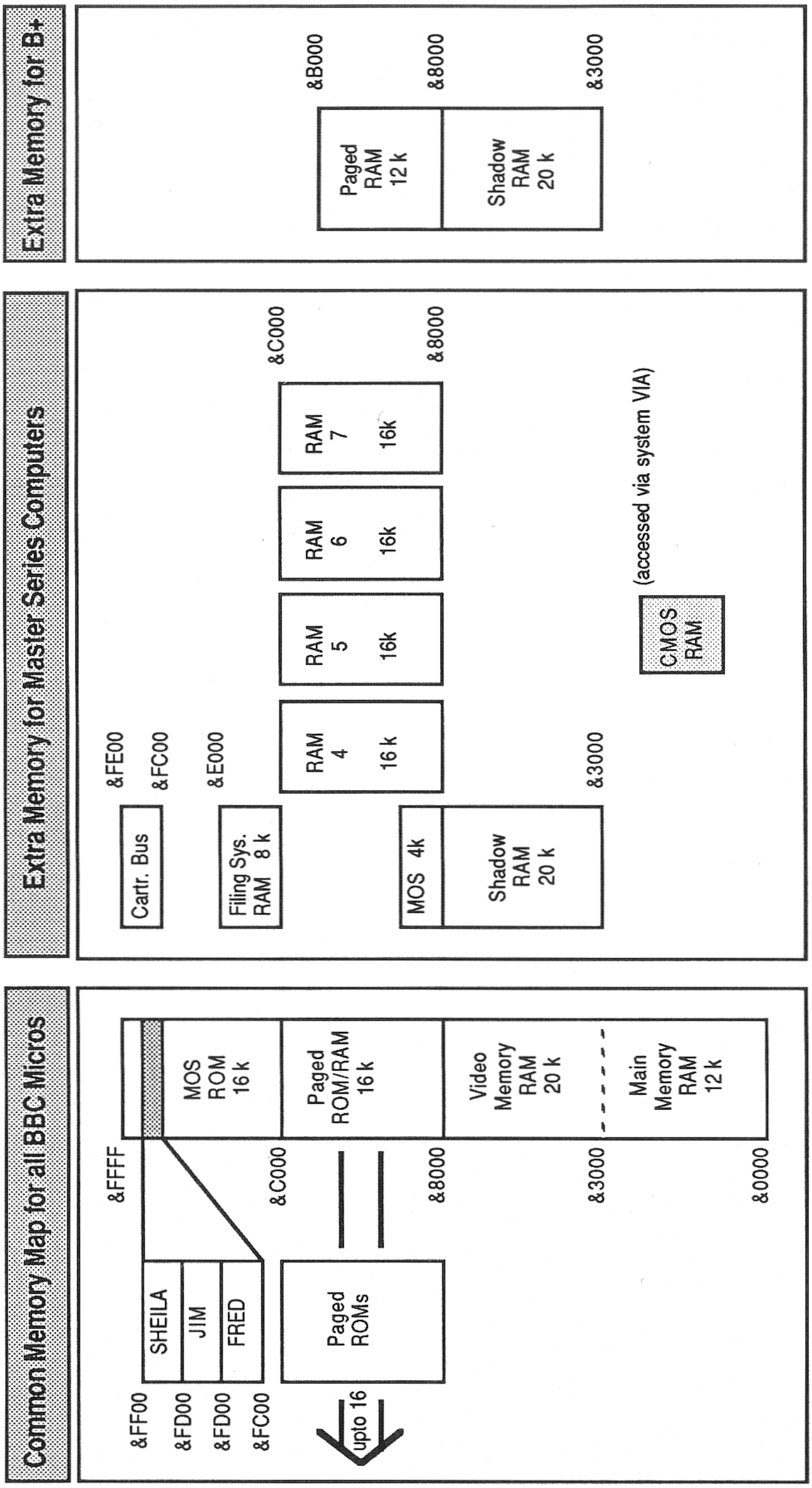


Fig 12.1 - The memory map

12.2 OSBYTE calls concerning memory use

The following OBYTE calls are relevant to the use of memory.

Read top of user memory (HIMEM) OSBYTE

Call address &FFF4
Indirected through &20A
A=&84 (132)

This call returns either the address of the bottom of the display RAM or the value &8000 if a shadow screen mode is in use. In a second processor this value is not the equivalent of HIMEM.

The address is returned in the X (LSB) and Y (MSB) registers

Read top of user memory for a given screen mode OSBYTE

Call address &FFF4
Indirected through &20A
A=&85 (133)

This call returns either the address of the bottom of the display RAM or the value &8000 if a shadow screen mode is selected.

Entry paramter:
X contains screen MODE number

On exit,
X contains LSB of address
Y contains MSB of address

Select screen memory for direct access OSBYTE

Call address &FFF4
Indirected through &20A
A=&6C (108)

This call is only available on Master series machines. It enables the selection of one of the two banks of memory from &3000-&7FFF to be selected as directly accessable to the user's program.

Entry parameters:
X=0 main memory selected
X=1 shadow memory selected

On exit:

X is preserved, Y is undefined

Select main/shadow memory for VDU access OSBYTE

Call address &FFF4
Indirected through &20A
A=&70 (112)

Only available on Master series machines, this call determines which memory is used by the VDU driver regardless of the memory displayed by the video hardware. Together with OSBYTE &71 rapid animation techniques may be developed by updating the next screen while the previous screen is being displayed.

Entry parameters:

X=0 main or shadow memory according to current mode (ie. main for modes 0-7 and shadow for modes 128-135).
X=1 main memory used
X=2 shadow memory used
Y=0

On exit:

X=old setting
C is undefined

Read/write flag used by OSBYTE &70

Call address &FFF4
Indirected through &20A
A=&FA (250)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

Available on Master series machines only.

Select main/shadow memory for display OSBYTE

Call address &FFF4
Indirected through &20A
A=&71 (113)

On Master series machines this call determines which memory is used by the video hardware to create the VDU display regardless of which memory the VDU driver is using.

Entry parameters:

X=0 main or shadow memory according to current mode
X=1 main memory displayed
X=2 shadow memory displayed
Y=0

On exit:

X=old setting
C is undefined

Read/write flag used by OSBYTE &71

Call address &FFF4
Indirected through &20A
A=&FB (251)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

Write shadow memory use OSBYTE

Call address &FFF4
Indirected through &20A
A=&72 (114)

On B+ and Master series machines this call can be used to force the use of shadow memory regardless of the screen mode selected.

Entry parameters:

X=0 shadow memory always used
X<>0 shadow memory not used if mode 0-7 selected
Y=0

On exit:

X=old setting
C is undefined

Test for sideways RAM OSBYTE

Call address &FFF4
Indirected through &20A
A=&44 (68)

This call is only available on Master series machines and the B+ and is used to detect the presence of sideways RAM. The operating system

tests that it is able to write a byte of memory in page &80 after switching in the memory bank using the latch at &FE30.

Entry parameters:
A=&44

On exit:
The value returned in X indicates which banks of sideways RAM are present.

bit 0 is set if ROM number 4 is RAM
bit 1 is set if ROM number 5 is RAM
bit 2 is set if ROM number 6 is RAM
bit 3 is set if ROM number 7 is RAM

Sideways RAM allocation OSBYTE

Call address &FFF4
Indirected through &20A
A=&45 (69)

This call is only available on Master series machines and the B+ and is used to return information about the use of sideways RAM. Sideways RAM banks may be allocated for use as paged ROMs or for use as extended memory using pseudo addresses.

Entry parameters:
A=&45

On exit:
The value in X indicates how banks of sideways RAM are being used.

bit 0 is set if ROM number 4 in use for extended addressing
bit 1 is set if ROM number 5 in use for extended addressing
bit 2 is set if ROM number 6 in use for extended addressing
bit 3 is set if ROM number 7 in use for extended addressing

RAM size OSBYTE

Call address &FFF4
Indirected through &20A
A=&FE (254)

This call has a different use on Master series computers. On a BBC model A or B this call returns information about the amount of memory installed.

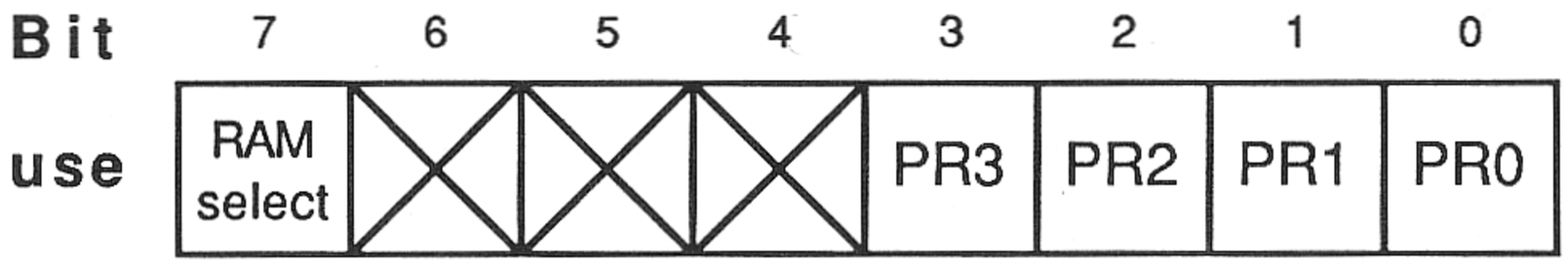
<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

On exit:

X=0 on the Electron
X=1 on the B+
X=&40 on a model A
X=&80 on a model B

12.3 Paged ROM and RAM hardware control

&FE30



The ROM paging register on all machines is accessed at 'Sheila' address &FE30. The lower 4 bits PR0-PR3 select which one of the 16 possible paged ROMs (and RAMs) is resident in the memory map at &8000 - &BFFF. The operating system keeps track of which paged ROM is being used at any time, so it will change the register contents quite often. Writing directly to this register is definitely not advised, especially from BASIC or any other language; the current language could be switched out of memory by mistake causing the whole machine to crash! Selection of ROMs should always be left to the MOS.

The Master computers set bit 7 of the paging register to switch a 4K bank of RAM into the memory map from &8000 - &8FFF. This segment of RAM is used by the MOS during graphics operations. On the B Plus bit 7 selects 12K of RAM from &8000 - &AFFF. It is possible to load ROMs into this area of RAM provided that they are less than 12K in length. ROM data loaded in this way may be selected as the current language using *FX142,128. Note however that any ROMs selected like this will not be retained as the current language over a BREAK.

Details of the standard configurations of paged RAM and ROM and full installation information is provided in section 17.2.

12.4 RAM Access Control

From the memory map diagram, it will be appreciated that fairly sophisticated control must be exercised over memory accesses, especially on the Master and B Plus. It is important to ensure that the

correct memory bank is switched into the correct location before trying to read or write data to that memory if the system is not to become an inoperational mess. Luckily, this control of the memory is normally handled by the MOS and the programmer does not need to worry about switching banks of memory himself. However, for the systems programmer writing filing systems or needing to exercise particularly tight control over the memory, it can be necessary to switch segments in and out of the main memory map. The remainder of this section covers this by describing the operations of the 'access control register' ACCCON which is located at Sheila address &FE34.

&FE34 - B plus

Bit	7	6	5	4	3	2	1	0
use	shadow select							

&FE34 - Master

Bit	7	6	5	4	3	2	1	0
use	IRR	TST	IFJ	ITU	Y	X	E	D

0	use shadow RAM for screen
1	use main memory for screen

0	VDU driver uses shadow RAM
1	VDU driver uses main memory

0	normal RAM in main memory
1	shadow RAM in main memory

0	8k RAM at &C000 - &DFFF
1	VDU driver code at &C000

0	enable external Tube
1	enable internal Tube

0	1MHz bus at &FC00-&FDFF
1	cartridge at &FC00

0	normal state (do not change)
1	hardware test

0	after IRQ processed
1	IRQ to CPU

12.4.1 Access to the Shadow RAM

With the B Plus came the introduction of the 'shadow' RAM. This is 20K of RAM which can be used to store the display data for the screen. When selected using *shadow, OSBYTE 114 or mode 128 - 135, the area of RAM in the main memory map from &3000 to &7FFF is freed for user programs and data. Control through the MOS in this manner is perfectly adequate for most applications. However, with the Master came more sophisticated access control to allow animated games and other applications which write data directly into the screen memory and shadow memory to be produced.

For the Master, several alternative approaches to direct screen control are possible, but it should always be born in mind that programs which write directly into the screen memory will not work from second processors. The first and simplest approach is to deselect the shadow screen altogether. Writing data into the screen display area is then just a matter of writing data to the correct address in memory without bothering about switching the shadow memory. This approach limits the RAM available for the user program, but does ensure compatibility with the model B. If extra RAM is required, the programmer will have to switch the shadow RAM into and out of the main map using the OSBYTEs covered in section 12.2. The code which performs this switching must not be in the memory from &3000 - &7FFF, otherwise it will switch itself out of the memory map and crash.

The Master series computers require the 'X' bit to be set to switch the 20K shadow RAM into the memory map. Clearing the 'X' bit switches the shadow RAM out of the memory map. Additional control is achieved with the 'E' and 'D' bits. With the 'E' bit set, any access to the screen memory addresses from the MOS VDU driver code at &C000-&DFFF automatically switches in the shadow RAM when data is written. With 'E' clear, all written data, even from the MOS VDU driver code, accesses the main memory. The 'D' bit determines whether the shadow or main memory is used by the 6845 CRTC to generate the video image. With 'D' set the shadow RAM is displayed; with 'D' clear the main memory is displayed. By careful control of all these bits, preferably using the OSBYTEs covered in section 12.2, smoothly animated sequences can be constructed. Generated video output is sourced from one of the screen memories, whilst the other is being updated with new graphics.

The B Plus does not have such versatile control of the shadow RAM. Its 'S' bit is equivalent to the Master's 'D' and 'E' bits combined. When set, the 'S' bit causes the VDU driver to update the shadow screen and causes the shadow screen to be displayed. When clear, the main memory is used by the MOS VDU driver and is displayed on the video screen.

12.4.2 Filing system RAM control

On the model B, filing systems such as the Disc Filing System (DFS) allocate necessary workspace at the bottom of available RAM. This leads to several pages of memory being occupied from &0E00 (the exact amount depends on the particular filing system(s) installed). Just like having the screen in main memory this is undesirable as it reduces the RAM available to the user.

The Master computers overcome this waste of memory by allowing an extra 8K bank of RAM to be switched in at &C000-&DFFF. This extra bank can be switched in by filing systems and temporarily overlays the MOS VDU driver code. To switch in the extra RAM, the 'Y' bit of ACCCON should be set, but must be cleared again before the VDU driver code next needs to be accessed.

12.4.3 Other Master access controls

In addition to RAM access control, the Master ACCCON register also performs the following functions:

Interrupts - setting the IRR bit causes an IRQ to the CPU

Cartridge/bus access - setting the IFJ bit causes accesses to 'Fred' and 'Jim' between &FC00-&FDFF to be directed to the cartridge sockets. When clear IFJ directs accesses to the 1MHz bus as on the model B.

Co-processors - the master can be attached to two secondary processors, one internal, the other external. When set, the ITU bit selects the internal co-processor and when clear it selects the external co-processor.

13 Video/Graphics System

The video system is described in two parts. All operating system calls and associated software support are described in the first section. The video hardware is described in the second section.

The manipulation of the video memory is described in the Memory chapter where the appropriate operating system calls and hardware are described.

13.1 O.S. Video Support

Screen output is achieved using the OS output routines with the screen selected as one of the current output streams. The OS output routines are described in chapter 6. The primary output routine is:

OSWRCH: Output character given in A
Call address &FFEE
Indirected through &20E

Using this routine all the defined character and graphics facilities are available. Most programmers will be familiar with these facilities when using BASIC.

13.1.1 Text output and control characters

When ASCII character values are output to the screen the appropriate character will be printed on the screen at the text cursor position. The ASCII values and characters are given in the following table.

hex.	dec.	ascii
20	32	space
21	33	!
22	34	"
23	35	#
24	36	$
25	37	%
26	38	&
27	39	'
28	40	(
29	41	)
2A	42	*
2B	43	+
2C	44	,
2D	45	-
2E	46	.
2F	47	/

hex.	dec.	ascii
30	48	0
31	49	1
32	50	2
33	51	3
34	52	4
35	53	5
36	54	6
37	55	7
38	56	8
39	57	9
3A	58	:
3B	59	;
3C	60	<
3D	61	=
3E	62	>
3F	63	?

hex.	dec.	ascii
40	64	@
41	65	A
42	66	B
43	67	C
44	68	D
45	69	E
46	70	F
47	71	G
48	72	H
49	73	I
4A	74	J
4B	75	K
4C	76	L
4D	77	M
4E	78	N
4F	79	O

50	80	P
51	81	Q
52	82	R
53	83	S
54	84	T
55	85	U
56	86	V
57	87	W
58	88	X
59	89	Y
5A	90	Z
5B	91	[
5C	92	\
5D	93	]
5E	94	^
5F	95	_

60	96	`
61	97	a
62	98	b
63	99	c
64	100	d
65	101	e
66	102	f
67	103	g
68	104	h
69	105	i
6A	106	j
6B	107	k
6C	108	l
6D	109	m
6E	110	n
6F	111	o

70	112	p
71	113	q
72	114	r
73	115	s
74	116	t
75	117	u
76	118	v
77	119	w
78	120	x
79	121	y
7A	122	z
7B	123	{
7C	124	|
7D	125	}
7E	126	~
7F	127	del

Character values between 0 and 32 are used to produce miscellaneous control effects such as moving the text cursor and performing PLOT commands. The table below gives a summary of these codes. A more detailed description can be found in the model B 'User Guide' or the 'Master Reference Manual part 1'.

Where a control character requires one or more parameters (indicated in the '+bytes' field of the table) these are taken as being the characters output immediately following the control character.

dec	hex	ctrl	+ bytes	description of function
0	0	@	0	Does nothing
1	1	A	1	Send next character to printer only
2	2	B	0	Enable printer
3	3	C	0	Disable printer
4	4	D	0	Write text at text cursor
5	5	E	0	Write text at graphics cursor
6	6	F	0	Enable VDU drivers
7	7	G	0	Make short beep (BEL)
8	8	H	0	Move cursor back one character
9	9	I	0	Move cursor forward one character
10	A	J	0	Move cursor down one line
11	B	K	0	Move cursor up one line
12	C	L	0	Clear text area
13	D	M	0	Carriage return
14	E	N	0	Paged mode on
15	F	O	0	Paped mode off
16	10	P	0	Clear graphics area
17	11	Q	1	Define text colour
18	12	R	2	Define graphics colour
19	13	S	5	Define logical colour
20	14	T	0	Restore default logical colours
21	15	U	0	Disable VDU drivers or delete input line
22	16	V	1	Select screen MODE
23	17	W	9	Re-program display character + various other fn's
24	18	X	8	Define graphics window
25	19	Y	5	PLOT k,x,y
26	1A	Z	0	Restore default windows
27	1B	[	0	ESCAPE value
28	1C	\	4	Define text window
29	1D	]	4	Define graphics origin
30	1E	^	0	Home text cursor to top left of window
31	1F	_	2	Move text cursor to x, y
127	7F	del	0	Backspace and delete

VDU code 23 may also be used to access the 6845 CRTC chip registers on the BBC microcomputer and to turn the cursor on or off on the BBC microcomputer and the Electron.

VDU23,0,r,v,0,0,0,0,0,0 - places value v in register r of the 6845 (two byte values may be given to the VDU commands by using a semi-colon instead of a comma; this method is used in the VDU23,1 examples below).

VDU23,1,0,0;0;0;0 - hides cursor
VDU23;1,1,0;0;0;0 - display cursor
VDU23;1,2,0;0;0;0 - gives a steady cursor
VDU23;1,3,0;0;0;0 - gives a flashing cursor

This method of selecting a steady cursor is only implemented on the Electron. When the cursor is hidden cursor editing may still be

performed. On the BBC micro,but not the Electron, the flashing cursor will re-appear during cursor editing.

On the Electron the VDU23,0,10,32;0;0;0 method of turning the cursor off using the 6845 register directly is also implemented for compatibility.

A brief summary of VDU 23,n commands available on the Master series computers is shown in the table below.

n	VDU 23,n function
0	Write to 6845 registers
1	Turn cursor on/off
2-5	Set Extended Colour Fill (ECF) patterns
6	Set dotted lines pattern
7	Scroll window directly
8	Clear block
9	Set 1st flash time
10	Set 2nd flash time
11	Set default ECF patterns
12-15	Set simple ECF pattern
16	Cursor movement control
17-25	reserved
26	reserved for Communicator font changes
27	reserved for Acornsoft sprites
28-31	User program calls

13.1.2 Reserved VDU23 expansion

The VDU23 functions numbered 17-31 are passed to the VDUV vector at &226. See the table above for detailed current allocations. These functions can be recognised by the following conditions on entry to VDUV: C=1, A contains the VDU function code, ie first number following 23 in the sequence. All parameters following the 23 are held in ascending order starting at VDU variable location &1B.

13.1.3 Read/write VDU queue OSBYTE call

Call address &FFF4
Indirected through &20A
A=&DA (218)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old queue value is returned in X.

This contains the 2's complement negative number of bytes still required for the execution of a VDU command.

Writing 0 to this location can be a useful way of abandoning a VDU queue. Otherwise writing to this location is not recommended.

13.1.4 PLOT numbers

The graphics plotting routines are available using control code value &19 (25). This control code takes 5 parameter bytes. The 5 bytes correspond to the K,X and Y parameters described for the PLOT command in BASIC as described in the 'User Guide'. The plot number (K) is a single byte value while the X and Y co-ordinates are two byte values.

A summary of the plot numbers is given below:

No.	Description
0	Move relative to last point
1	Draw relative in current foreground colour
2	Draw relative in logical inverse colour
3	Draw relative in current background colour
4	Move absolute
5	Draw absolute in current foreground colour
6	Draw absolute in logical inverse colour
7	Draw absolute in current background colour

Higher PLOT numbers have other functions. The graphics cursor movements and plot colours follow the same pattern as the first 8 plot numbers.

No.	Brief Description
8-15	Last point in line omitted when inverted plotting used
16-23	Using dotted line
24-31	Dotted line, omitting last point
32-39	Solid line, first point omitted
40-47	Solid line, both end points omitted
48-55	Dotted line, initial point omitted
56-63	Dotted line, both end points omitted
64-71	Single point plotting
72-79	Horizontal line filling
80-87	Plot and fill triangle
88-95	Horizontal line blanking (right only)
96-103	Plot rectangle
104-111	Horizontal line filling
112-119	Plot parallelogram
120-127	Horizontal line fill (non-foreground)
128-135	Flood fill (non-background)
136-143	Flood fill (foreground)
144-151	Plot circle outline
152-159	Plot filled circle
160-167	Plot arc
168-175	Plot filled chord segment
176-183	Plot filled sector
184-191	Move or copy rectangle
192-199	Plot ellipse outline
200-207	Plot solid ellipse
208-231	reserved
232-239	reserved for Acornsoft sprites
240-255	user PLOT numbers

13.1.5 VDU extension vector

VDUV
Indirection address &226

This vector is called when the VDU drivers are presented with an unknown command or a known command in a non-graphics MODE. It has been implemented to provide a facility for extending the graphics software.

A VDU 23,n command with a value of n in the range 2 to 31 will cause a call to be made to this vector with the carry flag set and the accumulator containing the value n.in the BBC model B. On machines equipped with the graphics extension ROM such as the Master Series, values from 17 to 31 are passed to this vector although some values have been reserved for expansion software (see 13.1.2).

An unrecognised PLOT command or the use of a PLOT command in a non-graphics MODE will result in this call being made with the carry flag clear. The accumulator will contain the PLOT number used.

13.1.6 User defined characters

The BBC microcomputer and the Electron allow the user to define the pixel pattern of text characters outside the ASCII and control character range. On Master series computers the font is permanently expanded and a default set of character definitions may be used.

In the default state 32 characters may be defined by the user using the VDU 23 statement from BASIC (or the OSWRCH call in machine code). Refer to the 'User Guide' for more information about using the VDU 23 command. These characters use memory from &C00 to &CFF. Printing ASCII codes in the range 128 (&80) to 159 (&9F) will cause these user defined characters to be printed up (these characters will also be printed out for characters in the range &A0-&BF, &C0-&DF, &E0-&FF).

Read character definition OSWORD

Call address &FFF1
Indirected Through &20C
A=&0A
X and Y contain the address of a parameter block

The 8 bytes which define the 8 by 8 matrix of each character which can be displayed on the screen may be read using this call. The ASCII value of the character definition to be read should be placed in memory at the address stored in the X and Y registers. After the call the 8 byte definition is contained in the following 8 bytes.

XY +	0	Character required
	1	Top row of character definition
	2	Second row of character definition
	-	
	-	
	8	Bottom row of character definition

Exploding the character definition RAM

In the default state of non-Tube model Bs the character definition RAM is said to be imploded because no memory has been allocated for storing extra character patterns. On Master series computers part of the additional RAM is permanently assigned to storing character patterns and so the concept of *exploding the font* is not applicable.

If the character definition RAM is exploded then ASCII characters 128 (&80) to 159 (&9F) can be defined as before using VDU 23 and memory at &C00. Exploding the character set definitions enables the user to uniquely define characters 32 (&20) to 255 (&FF) in steps of 32 extra characters at a time. The operating system must allocate memory for this, which it does using memory starting at the "operating system high-water mark" (OSHWM). This is the value to which the BASIC variable PAGE is usually set and so if a totally exploded character set is to be used in BASIC then PAGE must be reset to OSHWM+&600 (i.e. PAGE=PAGE+&600).

ASCII characters 32 (&20) to 128 (&7F) are defined by memory within the operating system ROM when the character definitions are imploded.

Explode character definition RAM OSBYTE call - model B
Restore default font definitions - Master series

Call address &FFF4
Indirected through &20A
A=&14 (20)

It should be noted that in a Master series computer or when a second processor is active the default state is a fully exploded font allocation. In the Master this call takes no parameters and restores the default font definitions.

Entry parameters :
X value explodes/implodes memory allocation

See section 6.6.3 for details about reading OSHWM.

The memory allocation for ASCII codes in the expanded state is as follows:

Parameter	ASCII code	Memory allocation
X=0	&80 - &8F	&C00 - &CFF (imploded)
X=1	&A0 - &BF	OSHWM - OSHWM+&FF (+above)
X=2	&C0 - &DF	OSHWM+&100 - OSHWM+&1FF (+above)
X=3	&E0 - &FF	OSHWM+&200 - OSHWM+&2FF (+above)
X=4	&20 - &3F	OSHWM+&300 - OSHWM+&3FF (+above)
X=5	&40 - &5F	OSHWM+&400 - OSHWM+&4FF (+above)
X=6	&60 - &7F	OSHWM+&500 - OSHWM+&5FF (+above)

Before the OSHWM is changed during a font explosion a service call is made to the paged ROMs warning them of the impending change.

On exit:

A is preserved
X contains the new OSHWM (high byte)
Y and C are undefined

Read character explosion state OSBYTE call

Call address &FFF4
Indirected through &20A
A=&B6 (182)

This call is implemented on the BBC model B and the Electron. However use of this call is not recommended as this OSBYTE has been re-allocated in the Master series where this OSBYTE is used to read/write the printer ignore character.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The explosion state is returned in X.

This location contains the state of font explosion as set by OSBYTE call with A=&14/*FX 20.

Restore a group of default font definitions - Master.

Call address &FFF4
Indirected through &20A
A=&19 (25)

This call was unused on previous BBC operating systems and on the Master series computers allows either complete restoration of the default font or restorations of groups of 32 characters.

Entry parameters :

X value determines which characters are restored.

Parameter	Group of Character Codes	
X=0	32 - 255	(&20 - &FF)
X=1	32 - 63	(&20 - &3F)
X=2	64 - 95	(&40 - &5F)
X=3	96 - 127	(&60 - &7F)
X=4	128 - 159	(&80 - &9F)
X=5	160 191	(&A0 - &BF)
X=6	192 - 223	(&C0 - &DF)
X=7	224 - 255	(&E0 - &FF)

13.1.7 Reading information about the screen state

A number of calls are implemented to enable the programmer to interrogate the machine to determine the state of the graphics or text screen.

Read character and screen MODE OSBYTE call

Call address &FFF4
Indirected through &20A
A=&87 (135)

This call returns the character at the text cursor and the number of the current screen MODE.

On exit:

X contains character value
(0 if character not recognised)
Y contains graphics MODE number
(0-7 regardless of shadow RAM modes)

A is preserved
C is undefined

Read output cursor position OSBYTE call

Call address &FFF4
Indirected through &20A
A=&A5 (165)

This call is available on the Master series operating systems only. It returns the current text cursor position(POS and VPOS). During copying of text on-screen, the co-ordinates of the output cursor are returned.

On exit:

X contains horizontal character position
Y contains vertical character position

Read input cursor position OSBYTE call

Call address &FFF4
Indirected through &20A
A=&86 (134)

This call returns the current text cursor position(POS and VPOS). During copying of text on-screen, the co-ordinates of the input cursor are returned.

On exit:

X contains horizontal character position
Y contains vertical character position

Read pixel value OSWORD call

Call address &FFF1
Indirected through &20C
A=&9
X and Y contain the address of a parameter block

This routine returns the status of a screen pixel at a given pair of X and Y co-ordinates. A four byte parameter block is required and the result is contained in a fifth byte.

XY +	0	LSB of the X co-ordinate
	1	MSB of the X co-ordinate
	2	LSB of the Y co-ordinate
	3	MSB of the Y co-ordinate

On exit:

XY+4 contains the logical colour at the pixel
or &FF if the point specified was off screen.

Read previous graphics positions OSWORD call

Call address &FFF1
Indirected through &20C
A=&0D

The operating system keeps a record of the last two graphics cursor positions in order to perform triangle filling if requested. These cursor positions may be read using this call. X and Y should provide the address of 8 bytes of memory into which the data may be written.

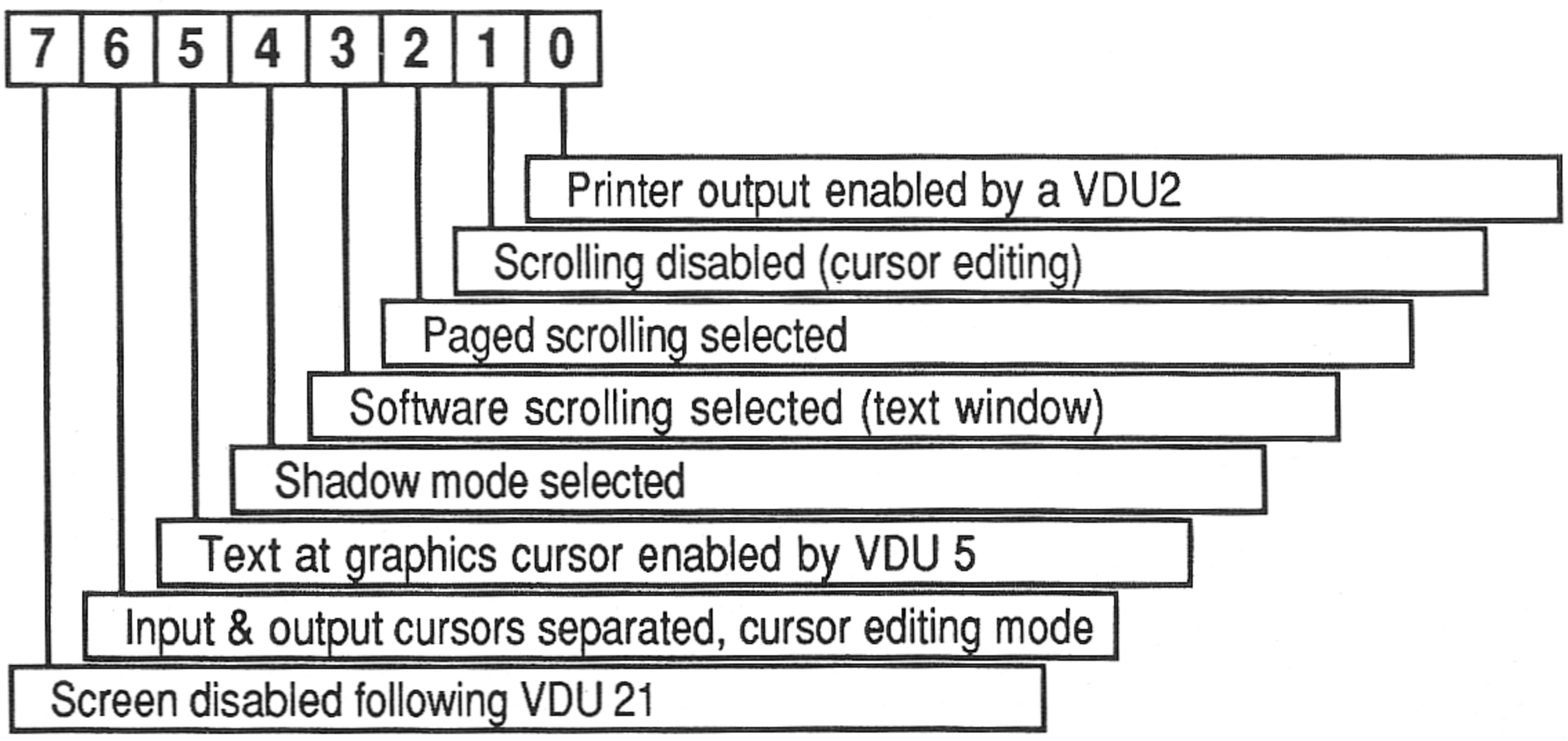
XY +	0	Previous X co-ordinate, low byte
	1	Previous X co-ordinate, high byte
	2	Previous Y co-ordinate, low byte
	3	Previous Y co-ordinate, high byte
	4	Current X co-ordinate, low byte
	5	Current X co-ordinate, high byte
	6	Current Y co-ordniate, low byte
	7	Current Y co-ordinate, high byte

Read VDU status OSBYTE call

Call address &FFF4
indirected through &20A
A=&75 (117)

On exit the X register contains the VDU status. Shadow mode (bit 4) is used only on the B+ and Master series machines. Information is conveyed in the following bits :

VDU Status Byte



A and Y are preserved
C is undefined

Read/write lines since last page halt OSBYTE

Call address &FFF4
Indirected through &20A
A=&D9 (217)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The previous setting of this location is returned in X.

This value represents the number of lines printed since the last page halt. It is used by the operating system to decide whether or not to halt scrolling when paged mode has been selected. This location is set to zero during OSWORD call &00 to prevent a scrolling halt occurring during input.

13.1.8 Reading and writing the colour states

The relationship between physical and logical colours, the palette, and the timings of the flashing colours may be accessed using the following calls.

Writing the palette

This may be achieved using the OSWRCH routine and control code &13 (19). A faster method uses the OSWORD call with A=&0C.

Write palette OSWORD call

Call address &FFF1
Indirection address &20A
A=&0C (12)
X and Y contain the address of a parameter block.

Another advantage of using this routine is that OSWORD calls can be used in interrupt routines. A parameter block should be set up with the logical colour being defined at XY, the physical colour being assigned to it in XY+1 and XY+2 to XY+4 containing padding 0's.

e.g. To perform an equivalent call to a VDU 19,1,3,0,0,0 command the parameter block should be:

XY +	0	Logical colour
	1	Physical colour
	2	0
	3	0
	4	0 (padding for future expansion)

Read palette OSWORD call

Call address &FFF1
Indirected through &20A
A=&0B (11)
X and Y contain the address of a parameter block

The physical colour associated with each logical colour may be read using this routine. On entry the logical colour is placed in the location at XY and the call returns with 4 bytes stored in the following four locations corresponding to a VDU 19 statement.

XY +	0	Logical colour
	1	Physical colour (on return)
	2	0
	3	0
	4	0 (padding for future expansion)

Set mark duration of flashing colours OSBYTE

Call address &FFF4
Indirected through &20C
A=&9 (9)

This call sets the duration of the mark state of flashing colours i.e. the duration of first named colour.

Entry parameters:

X determines length of duration, Y=0

X=0 Sets mark duration to infinity
Forces mark state if space is set to 0
X=n Sets mark duration to n centiseconds
(n=25 is the default setting)

On exit:

A is preserved
X contains the old mark duration
Y and C are undefined

Reading the mark duration OSBYTE call

Call address &FFF4
Indirected through &20A
A=&C2 (194)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old mark duration is returned in X.

Set space duration of flashing colours OSBYTE

Call address &FFF4
Indirected through &20C
A=&A (10)

This call sets the duration of the space state of flashing colours i.e. the duration of the second named colour.

Entry parameters:

X determines length of duration, Y=0

X=0 Sets space duration to infinity
Forces space state if mark is set to 0

X=n Sets space duration to n centiseconds
(n=25 is the default setting)

On exit:

A is preserved
X contains the old space duration
Y and C are undefined

Reading the space duration OSBYTE call

Call address &FFF4
Indirected through &20A
A=&C3 (195)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old space duration is returned in X.

Reading the flash counter OSBYTE call

Call address &FFF4
Indirected through &20A
A=&C1 (193)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flash counter value is returned in X. This counter contains the number of 1/50th sec. units until the next change of colour for flashing colours.

OSBYTE &9A (154)

Call address &FFF4
Indirected through &20A
A=&9A (154)

This call is implemented differently on the BBC microcomputer and the Electron.

(a) BBC Microcomputer and Master

Write to video ULA control register and OS copy

Entry parameters:
X contains value to be written

This call writes to register 0 of the video ULA and also writes the value into the reserved location of the operating system's workspace. See section 13.3.13.

This call also sets the flash counter to the mark value.

On exit:
A, X, Y and C are preserved

(b) Electron

Reset flash cycle

This call resets the flash cycle to the beginning of the mark state (i.e. to the first named colour of the pair) by manipulating the ULA registers.

There are no entry parameters.

On exit:

All registers are undefined

OSBYTE &9B (155)

Call address &FFF4
Indirected through &20A
A=&9B (155)

Write to video ULA palette register and OS copy

This call is not implemented on the Electron. On the Electron this call is ignored by immediately executing an RTS instruction.

Entry parameters:

X contains value to be written

This call writes to register 1 of the video ULA and also stores a copy of this value in the OS workspace. The actual value written to the register and the internal copy is X EOR 7.

On exit:

A, X and Y are preserved
C is undefined

OSBYTE &73 (115)

Call address &FFF4
Indirected through &20A
A=&73 (115)

Blank or restore palette (Electron)

This call is only implemented on the Electron where this call is used to set all colours to black or restore the default palette.

Entry parameters:

X =0, restore the palette
X>0, set palette colours to black

Reading OS copies of video ULA registers OSBYTE calls

Call address &FFF4
Indirected through &20A
A=&B8 (184) control register
A=&B9 (185) palette register

Read OS copies of video ULA registers.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

The last value written to the ULA registers can be read using this method. These calls should not be used to write to these locations because to do so would make the internal operating system copy of the registers inconsistent with the actual register contents. OSBYTE &B8 is undefined on the Electron and OSBYTE &B9 is used as the paged ROM polling semaphore (see chapter 17).

13.1.10 Wait for vertical sync OSBYTE call

Call address &FFF4
Indirected through &20A
A=&13 (19)

This call forces the machine to wait until the start of the next frame of the display. This occurs 50 times per second on the UK BBC Microcomputer. Its main use is to help produce flicker free animation on the screen. The flickering effect is often due to changes being made on the screen half way through a screen refresh. Using this OSBYTE call graphics manipulation can be made to coincide with the flyback between screen refreshes. User trapping of IRQ1 may stop this call from working.

On exit:

A is preserved
X, Y and C are undefined

13.1.11 The OS VDU variables

The operating system VDU variables are accessible on the BBC Microcomputer and Electron using the OSBYTE calls described below. These VDU variables have been officially described by Acorn in the 'Master Reference Manual Part 1'.

Read VDU variable value OSBYTE call

Call address &FFF4
Indirected through &20A
A=&A0 (160)

This call is used to read VDU variables and is preferable to reading directly from the video workspace.

Entry parameters:

X contains the number of the VDU variable number to be read

On exit:

X=low byte of number
Y=high byte of number
A is preserved
C is undefined

Read address of VDU variables OSBYTE calls

Call address &FFF4
Indirected through &20A
A=&AE (174) reads LSB
A=&AF (175) reads MSB

When used across the Tube the address returned refers to the i/o processor's memory.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The origin address LSB or MSB for the VDU variables is returned in X. With OSBYTE 175 the contents of the next location are returned in Y. This call returns with the address of the table of internal VDU variables.

On exit:

X=low byte
Y=high byte

13.1.12 *TV OSBYTE call

Call address &FFF4
Indirected through &20A
A=&90 (144)

This call may be used to set the vertical screen shift and interlace option as described for the *TV command.

Entry parameters:

X=vertical screen shift in lines
Y=0 interlace on
Y=1 interlace off

On exit:

X=old screen shift setting
Y=old interlace option

13.2 Video memory use

13.2.1 VDU zero page allocation

The allocation of zero page for the VDU driver is shown below.

add.	memory usage
&D0	STATS - VDU status byte
&D1	ZMASK - current graphics point mask
&D2	ZORA - text colour OR mask
&D3	ZEOR - text colour EOR mask
&D4	Graphics colour OR mask
&D5	Graphics colour EOR mask
&D6 &D7	Graphics character cell address
&D8 &D9	Text character cell address
&DA &DB &DC &DD &DE &DF	temporary VDU work space ZTEMP, ZTEMPA, ZTEMPB
&E0 &E1	Ptr to mult. tables (not Master series)

13.2.2 OS use of page 3

addr.	memory usage
&300 &307	Current graphics window, internal co-ordinates
&308 &30B	Current VDU 28 settings (current text window)
&30C &30F	Current VDU 29 settings (graphics origin)
&310 &313	Current graphics cursor
&314 &317	Old graphics cursor (internal co-ordinates)
&318	POS value
&319	VPOS value
&31A	POINT value within current graphics character
&31B &31E	VDU queue/Graphics workspace
&31F &323	VDU queue anchored at &323
&324 &327	Current graphics cursor (internal co-ordinates)
&328 &349	General graphics workspace
&34A &34B	Text cursor address for 6845
&34C &34D	Text window size in bytes (used for scrolling)
&34E	MSB of HIMEM regardless of shadow mode
&34F	No. of bytes occupied by a character for current mode
&350 &351	Address of top left of screen as used for 6845
&352 &353	No. of bytes used for row of characters for current mode
&354	Size of current screen mode in pages
&355	Current screen mode ignoring shadow bit
&356	Size of screen memory: 20k=0,16k=1,10k=2,8k=3,2k=4
&357 &35A	OS current colour settings
&35B &35C	VDU 18 settings, foreground plot mode
&35D &35E	Vector used when decoding VDU codes
&35F	Last setting of 6845 cursor start register
&360	(No. of logical colours for current mode)-1
&361	(No. of pixels per byte for current mode)-1, 0 for text
&362	Bit mask for right most pixel for current mode
&363	Bit mask for left most pixel for current mode
&364 &365	X+Y co-ordinates for text i/p cursor (used for copying)

addr.	memory usage
&366	Teletext output cursor character (&FF - default)
&367	Font flags, source is RAM if bit set
&368 &36E	Font location bytes, MSB of address for each font group
&36F &37E	Colour palette, 1 byte per logical colour
&37F	Unused byte
&380 &39C	Header block for BPUT file
&39D	Offset of next byte for BPUT o/p
&39E	Offset of next byte for BGET i/p
&39F &3A6	Unused bytes
&3A7 &3B1	Filename for the current BGET file
&3B2 &3D0	Block header of the most recent block read
&3D1	*OPT 3 value, sequential block gap
&3D2 &3DC	Filename of the current file being searched for
&3DD &3DE	Number for next BGET block
&3DF	Misc. CFS work space
&3E0 &3FF	Keyboard input buffer

addr.	memory usage
&366	VDU 23,16 setting
&367	VDU 23,6 setting, dot pattern
&368	Current state of dot pattern
&369	Colour plotting, ECF pattern number or zero
&36A	Graphics foreground, ECF number or zero
&36B	Graphics background, ECF number or zero
&36C	Bit 7 - flag for cursor in column 81
&36D	GCOL setting of graphics foreground colour
&36E	GCOL setting of graphics background colour

The diagram below describes the way in which the Acorn BBC series computers use the VDU driver workspace in page 3 of memory. Some *official* information can be found in the Master reference manuals. The remainder is unofficial and should be used with some caution.

13.3 The Video hardware

The video system on the Acorn-BBC range of computers is based around two core chips; the 6845 CRT chip and the Acorn Video ULA. The 6845 controls the general screen format, vertical cursor size and copes with light pens while the special Acorn Video ULA controls the colours and horizontal cursor size.

For most applications the operating system provides an extensive range of both character and graphics display facilities, which should be used wherever possible. Direct control of both the video chips and the video display memory can be used to obtain specialised effects like rapid sideways scrolling and fast animated graphics. This section covers the detailed information required to achieve this direct control.

13.3.1 Direct access to the display memory

Filling the screen memory directly with display data is not recommended for software which must operate across the entire Acorn-BBC range of computers since it will not work across the Tube. However, fast animated graphics applications (like some games) are only likely be able to achieve the desired speed using this method.

There are three essential points which must be considered before a program starts to write data into the screen memory:

(1) Is this program running from a second processor? If it is you cannot write directly into the screen memory because it is not in the memory map of your processor. Note that you can still access the video controller chips using valid OS calls.

(2) Is the Shadow Screen being used? Users of the B Plus and Master series may be using the shadow screen. If this is the case, it is necessary to switch the screen into the main memory map whenever it needs updating. TAKE CARE! The OS normally looks after switching this memory in and out. You MUST ensure that your code is not running from a part of memory which will be switched out of the memory map (if this happens your code will switch itself out of memory and the machine will crash). See section 12.4 for details of memory switching. Remember to switch the original non-shadow memory back in after writing into the shadow screen.

Be careful not to confuse 'selection' of the shadow screen with 'switching' the shadow screen using the ACCCON register. It is perfectly valid to select a shadow screen using *shadow. This tells the OS to write all video output into the shadow screen, but it only switches

the shadow screen into the memory map momentarily whenever it writes a character to it.

(3) Is the screen memory in the main memory map? If your machine is a model B, this will always be the case. You can force a B Plus or Master series model to use main memory by de-selecting the shadow screen. Once this has been done, you can write directly into the screen memory at its normal memory-mapped location (see section 13.4 for full details of these memory maps).

13.3.2 The 6845 CRTC

Sheila address &FE00-&FE01

General introduction to the 6845

The 6845 cathode ray tube controller chip (CRTC) forms the heart of the Acorn-BBC series video display circuitry. Its major function is that of displaying the video data in memory on a raster scan display device (a television or monitor). The 6845 is responsible for producing the correct format on the display device, positioning the cursor, performing interlace if it is required and monitoring the light pen input. It will use the main memory or the shadow memory for video display, as defined by the ACCCON register (see section 12.4).

From a user's point of view it is useful to know how to define a specialised screen layout, and how the screen layouts (modes) have been defined by Acorn. A generalised overview of the 6845 is therefore given first, followed by the values in each of the registers in the various modes. Section 13.4 contains diagrams illustrating all of the screen modes in a very concise and easily referenceded format.

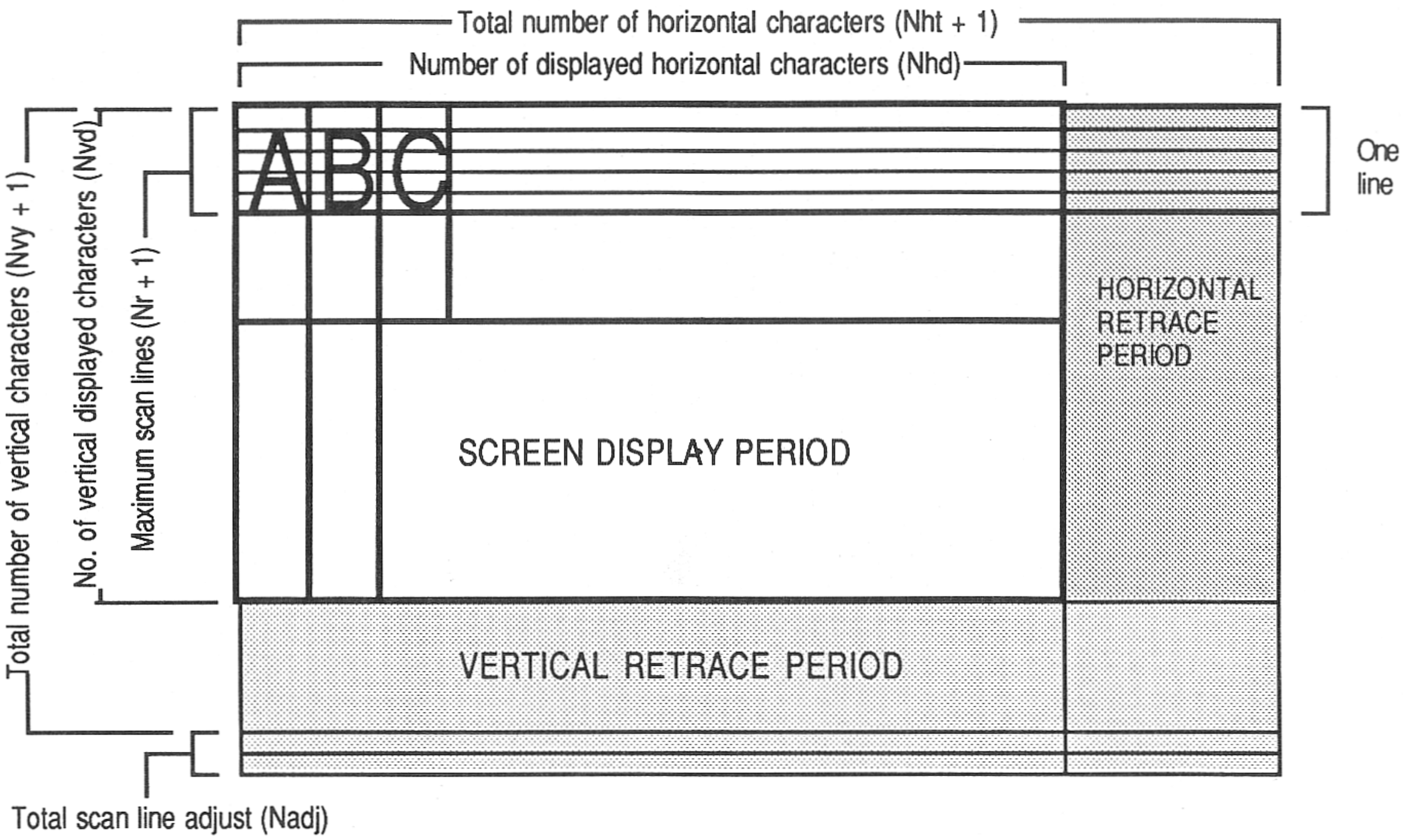


Fig 13.1 - Illustration of a general screen format

13.3.3 Programming the 6845

The 6845 possesses 18 internal registers, 14 of which are write only (R0-R13), 2 of which are read and write (R14-R15) and 2 of which are read only (R16-R17). In order to gain access to any of these registers, the register address must first be written into the 6845 address register. This is situated at Sheila address &FE00. Having written a 5 bit number into the address register, the selected internal register may be written to, or read from, at Sheila address &FE01.

The best way of programming the 6845 is by using the VDU23 command because this will work across the Tube. For example VDU23,0,R,V,0,0,0,0,0,0 will put the value V into register R. In BASIC programs it can be shortened by using semicolons instead of commas. A semicolon causes a 2 byte word to be included in the VDU command. For example VDU23;R,V;0;0;0 has the same effect as the first example.

6845 REGISTER SUMMARY TABLE

6845 register	Function	Contents in mode:							
		0	1	2	3	4	5	6	7
R0	Horizontal total	127	127	127	127	63	63	63	63
R1	Characters per line	80	80	80	80	40	40	40	40
R2	Horizontal sync. position	98	98	98	98	49	49	49	51
R3	Horizontal sync. width	8	8	8	8	4	4	4	4
	Vertical sync. time	2	2	2	2	2	2	2	2
R4	Vertical total	38	38	38	30	38	38	30	30
R5	Vertical total adjust	0	0	0	2	0	0	2	2
R6	Vertical displayed characters	32	32	32	25	32	32	25	25
R7	Vertical sync. position	34	34	34	27	34	34	27	27
R8	Interlace mode bits 0,1	1	1	1	1	1	1	1	1
	Display delay bits 4,5	0	0	0	0	0	0	0	1
	Cursor delay bits 6,7	0	0	0	0	0	0	0	2
R9	Scan lines per character	7	7	7	9	7	7	9	18
R10	Cursor start bits 0-4	7	7	7	7	7	7	7	18
	Cursor type bit 5	1	1	1	1	1	1	1	1
	Cursor blink bit 6	1	1	1	1	1	1	1	1
R11	Cursor end	8	8	8	9	8	8	9	19
R12,R13	Screen start address								
R14,R15	Cursor position								
R16,R17	Light pen position								

= variable values

13.3.4 The Horizontal Timing Registers

The horizontal timing registers define all of the horizontal timing for the screen layout. The point of reference for these registers is the left most displayed character position. The registers are programmed in "character time units" relative to the reference point.

Horizontal total register (R0)

This 8 bit write only register determines the horizontal sync. frequency. It should be programmed with the total number of displayed plus non-

displayed character time units across the screen minus one (Nht on figure 13.1).

Note that the number of displayed characters is not necessarily the same as the number of characters per line. This is because of the variable number of bits attributed to each pixel, which depends upon the number of colours available. The table for R1 contents illustrates this.

Mode	0	1	2	3	4	5	6	7
R0	127	127	127	127	63	63	63	63

Horizontal displayed register (R1)

This 8 bit write only register determines the number of displayed characters per horizontal line (Nhd on figure 13.1).

Mode	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
No. of CHRS as seen by 6845 (Nhd)	80	80	80	80	40	40	40	40
No. of CHRS as seen on the screen	80	40	20	80	40	20	40	40
No. of bits used to store colour information	1	2	4	1	1	2	1	1

Horizontal sync. position register (R2)

This 8 bit write only register determines the horizontal sync. pulse position on the horizontal line. The specification is in terms of character widths from the left hand side of the screen.

Mode	0	1	2	3	4	5	6	7
R2	98	98	98	98	49	49	49	51

Increasing the value of this register pushes the entire screen left whilst decrementing it pushes the whole screen to the right.

The sync. width register (R3)

This 8 bit write only register defines both the horizontal and the vertical sync. pulse times.

Horizontal sync. pulse width

The lower 4 bits contain the horizontal sync. pulse width in number of characters. Any number between 1 and 15 can be programmed, but 0 is not valid. It is however not advisable to change this register since most monitors and televisions require the standard sync. width to operate properly.

Mode	0	1	2	3	4	5	6	7
Lower 4 bits of R3	8	8	8	8	4	4	4	4

Vertical sync. pulse width

The upper 4 bits contain the number of scan line times for the vertical sync. pulse. This is set to 2 in all modes.

13.3.5 The Vertical Timing Registers

The point of reference for vertical registers is the top displayed character position. Vertical registers are programmed in character row times or scan line times.

Vertical total register (R4)

The vertical sync. frequency is determined by both R4 and R5. In order to obtain an exact 50Hz or 60Hz vertical refresh rate, the required number of character line times is usually an integer plus a fraction. The integer number of character lines minus one (Nvt on figure 13.1) is programmed into this 7 bit write only register.

Mode	0	1	2	3	4	5	6	7
R4	38	38	38	30	38	38	30	30

Vertical total adjust register (R5)

This 5 bit write only register is programmed with the fraction for use in conjunction with register R4. It is programmed with a number of scan lines (Nadj on figure 13.1). It can be varied slightly in conjunction with R4 to move the whole display area up or down a little on the screen. It is usually set to 0 except when using modes 3,6 and 7 in which it is set to 2. *TV (OSBYTE &90) controls the vertical positioning of a display on the screen. Refer to section 13.1.12.

Vertical displayed register (R6)

This 7 bit write only register determines the number of displayed Character rows (Nvd on figure 13.1) on the CRT screen and is programmed in character row times.

Mode	0	1	2	3	4	5	6	7
Character lines	32	32	32	25	32	32	25	25

Vertical sync. position (R7)

This 7 bit write only register determines the vertical sync position with respect to the reference. It is programmed in character row times.

Mode	0	1	2	3	4	5	6	7
Sync position	34	34	34	27	34	34	27	27

13.3.6 Interlace and delay register (R8)

This 6 bit write only register controls the raster scan mode and cursor/display delay.The interlace options are:

Interlace modes (bits 0,1)

Interlace mode register	Description
Bit 1 Bit 0	
0 0	Normal (non-interlaced) sync. mode (figure 13.2a)
1 0	Normal (non-interlaced) sync. mode (figure 13.2a)
0 1	Interlace sync mode (figure 13.2b)
1 1	Interlace sync and video (figure 13.2c)

All BBC microcomputer screen modes are interlaced sync only except for mode 7 which is interlaced sync and video. The default values can easily be changed using *TV (*FX 144) followed by a 0 to turn interlacing on or a 1 to turn interlacing off.

Fig. 13.2a Normal Sync.

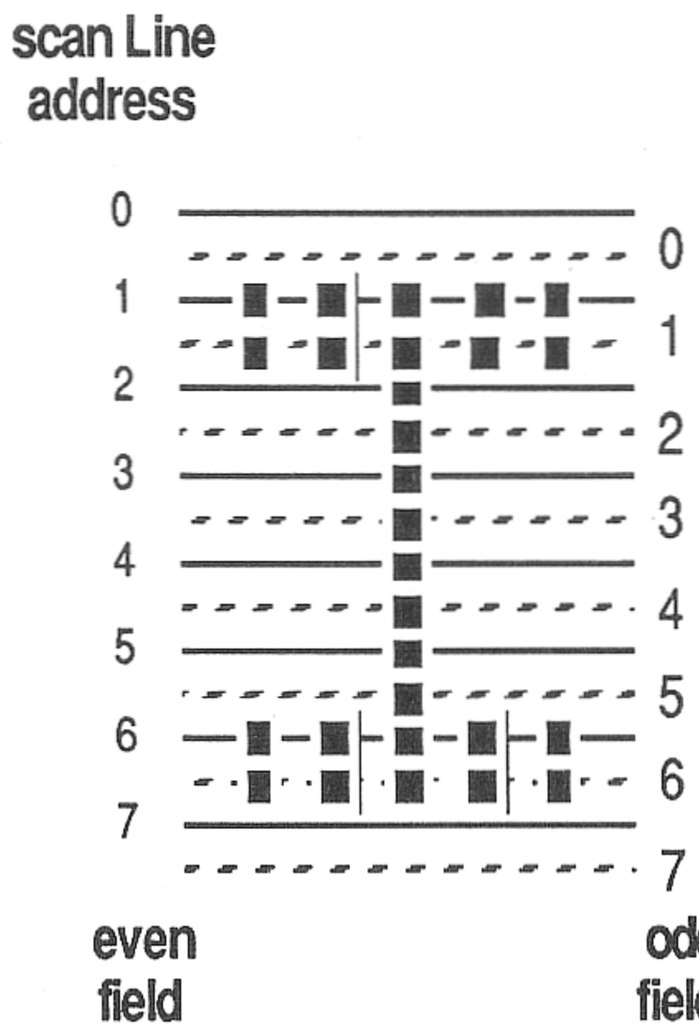


Fig. 13.2b Interlace Sync.

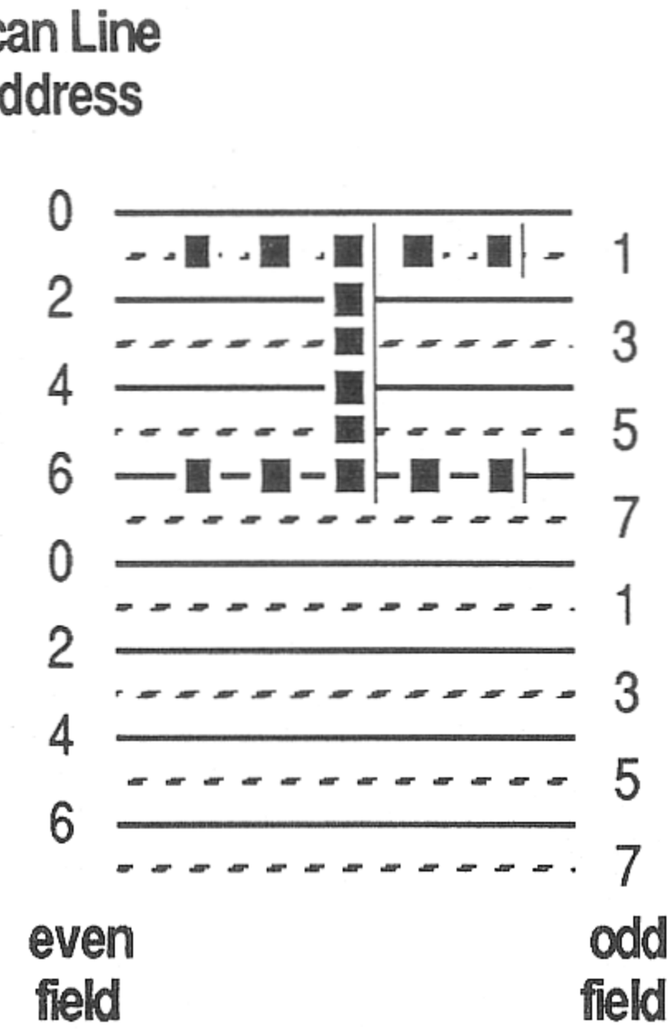


Fig. 13.2c Interlace Sync. & Video

Display blanking delay (bits 4,5)

Bits 4 and 5 control the display blanking signal. This signal must be enabled for all of the character output period and is used to take account of the time to transfer data from memory to the video output circuitry. No delay is required in modes 0-6, but a one character delay is required in mode 7 because the SAA5050 character generator is used.

Display blanking delay

Bit 5	Bit 4	Description
0	0	No delay
0	1	One character delay
1	0	Two character delay
1	1	Disable video output

Cursor blanking delay (bits 6,7)

Bits 6 and 7 control the cursor blanking signal. This signal must be enabled at the exact time when a cursor should appear on the screen. No delay is required in modes 0-6, but a two character delay is required in mode 7.

Cursor enable signal

Bit 7	Bit 6	Description
0	0	No delay
0	1	One character delay
1	0	Two character delay
1	1	Disable cursor output

13.3.7 Scan lines per character (R9)

This 5 bit write only register determines the number of scan lines per character row including spacing. The programmed value is one less than the total number of output scan lines.

Mode	0	1	2	3	4	5	6	7
Scans per character	7	7	7	9	7	7	9	18

13.3.8 The Cursor

It is possible to program a cursor to appear at any character position (defined by R14 and R15). Its blink rate can be set to 16 or 32 times the field period of 20 ms. Optional non-blink and non-display (i.e no cursor on the screen) modes can also be selected. The cursor height in number of lines and vertical position in character slots can also be defined.

The cursor start register (R10)

This 7 bit write only register controls the cursor format (see figure 13.3). Bit 7 is not used. Bit 6 enables or disables the blink feature. Bit 5 is the blink timing control bit. When bit 5=0, blink frequency = 1/16th of the field rate. When bit 5=1, blink frequency = 1/32nd of the field rate. The cursor start line is set by the lower five bits.

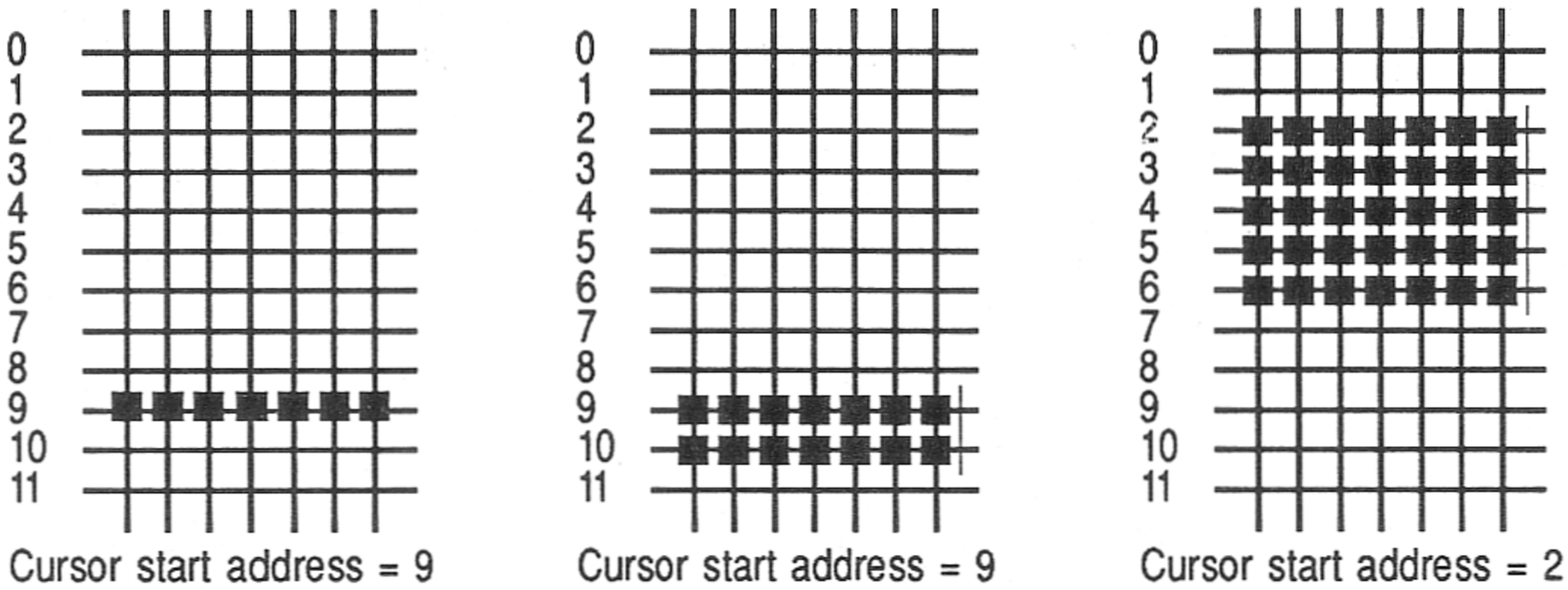


Figure 13.3 - Cursor layout examples

The cursor end register (R11)

This 5 bit write only register sets the cursor end scan line (see diagram).

Mode	0	1	2	3	4	5	6	7
Cursor end	8	8	8	9	8	8	9	19

Cursor position register (R14 and R15)

This 14 bit read/write register stores the current cursor location. It consists of 8 low order (R15) and six high order (R14) bits.

13.3.9 Light Pens

A typical light pen consists of a small light sensitive device fixed to the end of a pen-shaped holder. The sensor picks up the light given out from the monitor screen and sends an electronic signal into the micro. This LPSTB (light pen strobe) signal can be decoded (because the screen is scanned on a raster basis) and the position of the pen head determined.

Light pens can be used for a multitude of tasks such as drawing, "painting", designing layouts, playing games etc., but their use in many applications is limited by the resolution. The reason for this is that a fairly large area of screen (i.e. perhaps 5mm x 5mm) is ususally required to provide sufficient light to operate the pen. The maximum resolution for defining the position of the light pen is therefore a patch on the screen of this size, so accurate line drawings are impossible. The position of the light pen is stored to the nearest character position, so this limits the resolution to a character cell.

Light pen position register (R16 and R17)

This 14 bit read only register is used to store the location of a light pen sensor placed in front of the screen. The register is modified whenever the LPSTB signal is pulsed high.

Constructing a light pen

The light pen hardware must produce a positive going TTL pulse whenever the display scan position is under the sensor. The light pen position will then be stored in the light pen register R16,R17. Note that slow light pen response will require a delay factor to be subtracted from R16,R17 to produce the correct light pen position.

Luckily, there are small light sensitive devices available which provide a direct TTL logic level output. If one of these is fixed to the end of an empty pen and connected to the light pen input on the rear of the BBC microcomputer, an operational light pen can be constructed. The connections for such a pen are illustrated in figure 13.4. A special photosensor called a "Sweet spot" is available from R S Components or most of their distributors, and is supplied as part number RS 303-292.

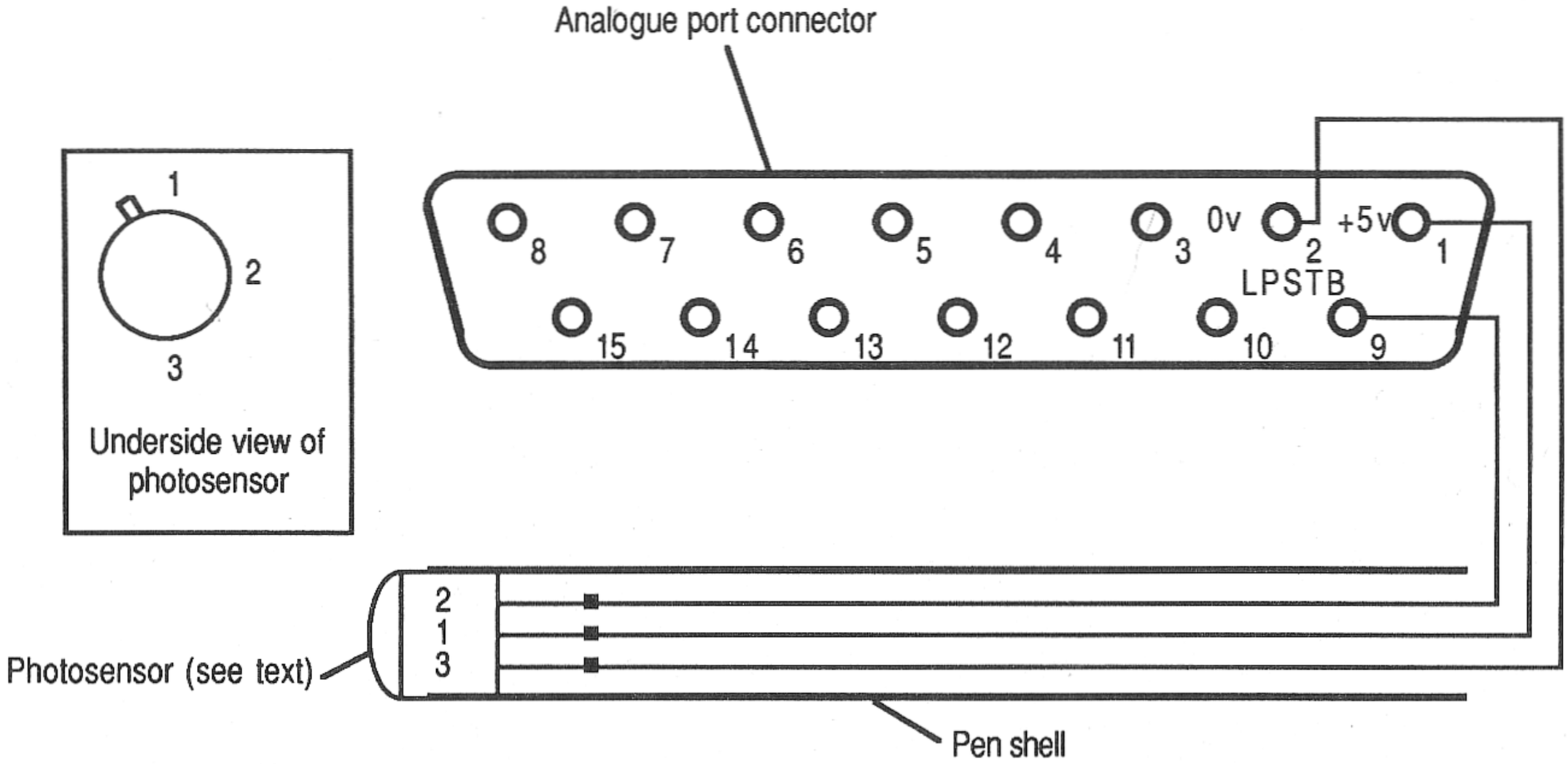


Figure 13.4 - Light pen circuit

Light pen software

In order to take account of the different screen start addresses for the various modes, a further correction factor must be subtracted from the contents of the light pen register. These correction factors are:

Mode	Correction factor
0	&0606 (1542)
1	&0606 (1542)
2	&0606 (1542)
3	&0806 (2054)
4	&0B04 (2820)
5	&0B04 (2820)
6	&0C04 (3076)
7	&2808 (10248)

The light pen position in terms of x,y coordinates is given by:

y = (L.p register-correction) DIV number of characters per line

x = (L.p register-correction) MOD number of characters per line

This x value will be in terms of 6845 characters and will have to be modified by multiplying by:

$$\frac{\text{Number of chars. per line}}{\text{Number of chars. seen by 6845}}$$

e.g. for mode 2 : $\frac{20}{80} = \frac{1}{4}$

The resolutions are therefore:

Modes 0,3,4,6,7 single displayed character
Modes 1,5 half of a displayed character
Mode 2 quarter of a displayed character

Note that the screen should be cleared before using a light pen and not scrolled whilst the pen is in use. If it is scrolled, the position of the start of the screen will have to be taken into account as well.

```
10 DIM M% 150
20 olp=&70:lpen=&74
30 FOR opt%=0 TO 3 STEP 3
40   P%=M%
50   [
60   OPT opt%
70   .init  SEI          \ Disable interrupts
```

```
 80           LDA #int MOD 256 \ Low byte of address
 90           STA &206    \ IRQ2V low
100           LDA #int DIV 256 \ High byte of address
110           STA &207    \ IRQ2V high
120           LDA #&88    \ Interrupt change mask
130           STA &FE4E   \ Enable lightpen interrupt
140           CLI
150           RTS         \ Exit
160    .int   LDA &FC     \ Do save ...
170           PHA
180           TXA
190           PHA
200           TYA
210           PHA
220           LDA &FE4D   \ Get system VIA interrupt status
230           AND #&88    \ Mask out bits not interested in
240           CMP #&88    \ Is it a lightpen interrupt?
250           BNE exit    \ No - exit
260           LDA &FE40   \ Clear interrupt
270           LDX #16     \ Lightpen register
280           STX &FE00   \ 6845 address
290           INX         \ Ready for next read
300           LDA &FE01   \ 6845 data
310           CMP olp+1   \ =old value?
320           STA olp+1   \ Update with new value
330           BNE diff1
340           STX &FE00   \ Next register
350           LDA &FE01   \ Get low address
360           TAY         \ Temporary store
370           SBC olp     \ Is it nearly eq.
380           CLC
390           ADC #1      \ Nearly eq if 0,1,2
400           BMI diff2
410           CMP #3      \ Compare with 2+1
420           BCS diff2   \ >=3 so not nearly eq..
430    \ Have two value s same so update lpen
440           STY lpen
450           LDA olp+1
460           STA lpen+1
470           JMP exit    \ And depart
480    .diff1 STX &FE00   \ Next register
490           LDY &FE01
500    .diff2 STY olp     \ Update olp
510           LDA #0      \ Mark lpen as invalid
520           STA lpen
530           STA lpen+1
540    .exit  PLA         \ Restore registers ...
550           TAY
560           PLA
570           TAX
580           PLA
590           STA &FC     \ Just in case it has changed
600           RTI
610    ]
620    NEXT opt%
630 REM Initialise workspace
640 !olp=0
650 !lpen=0
660 REM grab the vector
670 CALL init
680 REM grab lightpen interrupts
690 REM Using mask 11110111=247
```

```
700 *FX 233,247
710 REM Demonstrate action of lightpen interrupts
720 REM Refer to hardware section for adjustments etc.
730 REM Set up a text window to stop hardware scroll
740 VDU 28,0,23,39,0
750 REPEAT
760   IF !lpen = 0 THEN PRINT "Not valid":ELSE PRINT ~!lpen
770   UNTIL FALSE
```

13.3.10 Displayed screen start address register (R12, 13)

This 14 bit write only register determines the location in memory which corresponds to the upper left-hand character displayed on the screen. R13 (8 bits) is the low order address and R12 (6 bits) is the high order address. It can often be useful to know what the current contents of this register are. Unfortunately, being a write only register it is not possible to read the value directly. However, OSBYTE &A0 can be used to get these parameters from the operating system workspace in page &03. The start of screen address is stored in locations &350 and &351. If OSBYTE is called with &A0 with X=&50 then the contents of &350 will be returned in the X register and the contents of &351 will be returned in the Y register.

Note that the actual screen start address must be divided by 8 before being sent to R12,R13 because there are 8 lines per character (modes 0-6). In mode 7 a rather more complex correction has to be applied (see section 13.3.11 on mode 7 scrolling).

The ability to define the start of the screen to be anywhere in memory is very useful because it allows fast scrolling of the screen up, down, left and right. Provided that the start address is inside the screen memory of the mode being used, a hardware wrap around feature will also operate. Characters which would have scrolled off the top of the screen will therefore reappear at the bottom. The wrap around circuit simply detects whenever the 6845 tries to get video data from a ROM (an address above &7FFF), and adds an offset to that address. This has the effect of bringing the address back inside the video RAM. Since the screen sizes are different in the various modes, 2 bits on the SYSTEM VIA are used to define the length of the hardware scrolled screen, see section 22.3.

13.3.11 Hardware Scrolling

Scrolling the screen fast in any direction can be of immense use in a large number of applications. Text can be scrolled in word processing

applications, landscapes can be made to rush by in a horizontal direction (see games such as Acornsoft Planetoid). If it were not for the hardware scroll feature, it would be necessary to move every byte on the screen to perform a scroll. This is very time consuming for 20000 bytes and therefore slow. In order to make effective use of the hardware scrolling facilities available, it is necessary to understand both the advantages and the limitations which are imposed.

Modes 0-6 will now be analysed in detail followed by mode 7 which is slightly different.

Modes 0-6 vertical scrolling

In order to move the screen position upwards by one character line, it is necessary to increment the current start address register (R12,R13) by the number of characters per line. Remember that these are characters as produced by the 6845 and not as seen on the screen. There are 80 6845 characters per line in modes 0-3 and 40 characters per line in modes 4,5 and 6. The screen can be scrolled downwards by decrementing the screen start address register by the number of characters per line. Note that you should not normally allow the screen start address register to contain a value less than the *official* screen start address or greater than the *official* screen end address in the mode being used. If this occurs then areas of the main system memory will be displayed directly on the screen. This produces some interesting results, especially if zero page is displayed! Remember that the value put into R12,R13 is the actual memory address DIV 8. (See the example program in section 13.3.12.)

Sideways scrolling

The whole screen can be made to move left by one character (as seen by 6845) by incrementing the screen start register. It will move one character to the right by decrementing this register. Note that each character which moves off the left of the screen will appear on the next line up at the right of the screen. It is therefore necessary to move each of these characters down a line in software to maintain a true sideways scroll.

This scrolling technique is good for text, but may produce jumpy movements in graphics due to the limited resolution in the screen position. On a mode 0 screen, each sideways scroll moves the screen by 8 pixels. On a mode 2 screen, each 6845 character only represents 2 graphics pixels so a fairly effective hardware scroll can be used.

Mode 7 scrolling

Hardware scrolling in mode 7 is slightly more complex than in modes 0-6. To calculate the value to put into registers 12 and 13, first calculate the required start address in RAM (e.g &7C28). Take the high byte and subtract &74, then EOR the result with &20. This new value should be put into R12. R13 contains the low order address byte. A similar correction factor should be applied when calculating the cursor register contents.

13.3.12 Fast Animation

Fast animation using mode 2

Mode 2 has several advantages over all of the other modes for fast animation. It is for this reason, plus the fact that all 16 colours are available, that this mode is used in most fast graphics games. Provided that the programmer is prepared to put up with a 2 pixel at a time movement instead of a 1 pixel at a time movement, moving objects simplifies to moving complete bytes in memory. Consider for a moment the layout of each byte on a mode 2 screen.

pixel	P2d	P1d	P2c	P1c	P2b	P1b	P2a	P1a
bit	7	6	5	4	3	2	1	0

P1a-P1d are 4 bits defining the colour of pixel 1
P2a-P2d are 4 bits defining the colour of pixel 2

To move graphics sideways by one pixel involves extracting P1a-P1d from P2a-P2d. These removed bits must then be reinserted into the adjacent byte. This process is tricky and consumes a lot of processing time leading to very slow movement in all but the simplest of cases. It will be appreciated how much faster it is to simply move a byte (2 pixels) at a time from one memory location to another, which can be done very fast indeed.

Fast animation using mode 0

Unlike mode 2, fast animation moving a byte at a time in mode 0 moves 8 pixels. Animation moving 8 pixels at a time will generally produce very uneven motion. Since the packing of pixels in mode 0 assigns one bit per pixel, animation can be implemented by shifting all of the bits in a byte left or right by one position using a "ROR" or "ROL" instruction.

The bit which moves off the edge of one byte must be put into the adjacent byte. Since this has to be applied to all bytes on the screen, it is a slow process.

Wrap around

To ensure that the hardware wrap around feature operates correctly, the start of screen address must be kept within the screen boundaries. If it goes below the start of screen address then add the length of the screen to it. If it goes above the top of screen address then subtract the length of the screen from it.

e.g. in mode 0, the calculated start of screen address may be &8050. Since this is outside of the screen, it should be changed to &3050 by subtracting &5000, the screen size.

Hardware scroll example

The program listed below uses the hardware scroll facilities in mode 0. A line of text can be moved around the screen using the cursor keys. Note that as text moves off one side of the screen (sideways scroll), it reappears on the other side either one line up or one line down from its original position. If a true sideways scroll is required, it is necessary to move all of the bytes on the relevant side of the screen up or down by one character position. During motion of the line of text in a vertical direction, there will be brief flashes of another line on the screen. This is partially due to the delay in BASIC between setting register 12 and register 13 on the 6845, and also because the change occurs in the middle of a screen display. The flashing will be reduced in machine code programs which wait until the frame sync. period before changing any of the 6845 registers.

```
10 REM HARDWARE SCROLL EXAMPLE IN MODE 0
20 MODE0
30 START=&3000
40 PRINT"THIS TEXT CAN BE SCROLLED IN ANY DIRECTION USING THE
   CURSOR KEYS"
50 REM SET KEYS REPEAT RATE AND CURSOR KEYS TO GIVE 136 ETC.
60 *FX4,1
70 *FX12,3
80 REPEAT
90   A=INKEY(0)
100   IF A=136 THEN PROCMOVE(8)
110   IF A=137 THEN PROCMOVE(-8)
120   IF A=138 THEN PROCMOVE(-640)
130   IF A=139 THEN PROCMOVE(640)
140   UNTIL FALSE
150 DEF PROCMOVE(offset)
```

```
160 START=START+offset
170 REM IF ABOVE OFFICIAL START THEN SUBTRACT SCREEN LENGTH
180 IF START>=&8000 THEN START=START-&5000
190 REM IF BELOW OFFICIAL START ADDRESS, ADD SCREEN LENGTH
200 IF START<&3000 THEN START=START+&5000
210 REM MODIFY 6845 MEMORY START ADDRESS REGISTER
220 VDU23;12,START DIV 2048;0;0;0
230 VDU23;13,START MOD 2048 DIV 8;0;0;0
240 ENDPROC
```

13.3.13 The Video ULA

Sheila address &FE20-&FE21

The Video ULA is a special chip designed by Acorn to provide all the video timing for the rest of the system (including the 6845), to determine the relationship between logical and physical colours, to control the cursor width and to provide Red, Green and Blue (R G B) video outputs. This section explains how the ULA is programmed for the various display modes.

The Video Control Register SHEILA &FE20 (write only)

This 8 bit register controls which flashing colour is present at any one time, whether teletext is selected, the number of characters per line, the clock rate sent to the 6845 and the master cursor size. OSBYTE 154 should be used to write data into this register to maintain Tube compatibility.

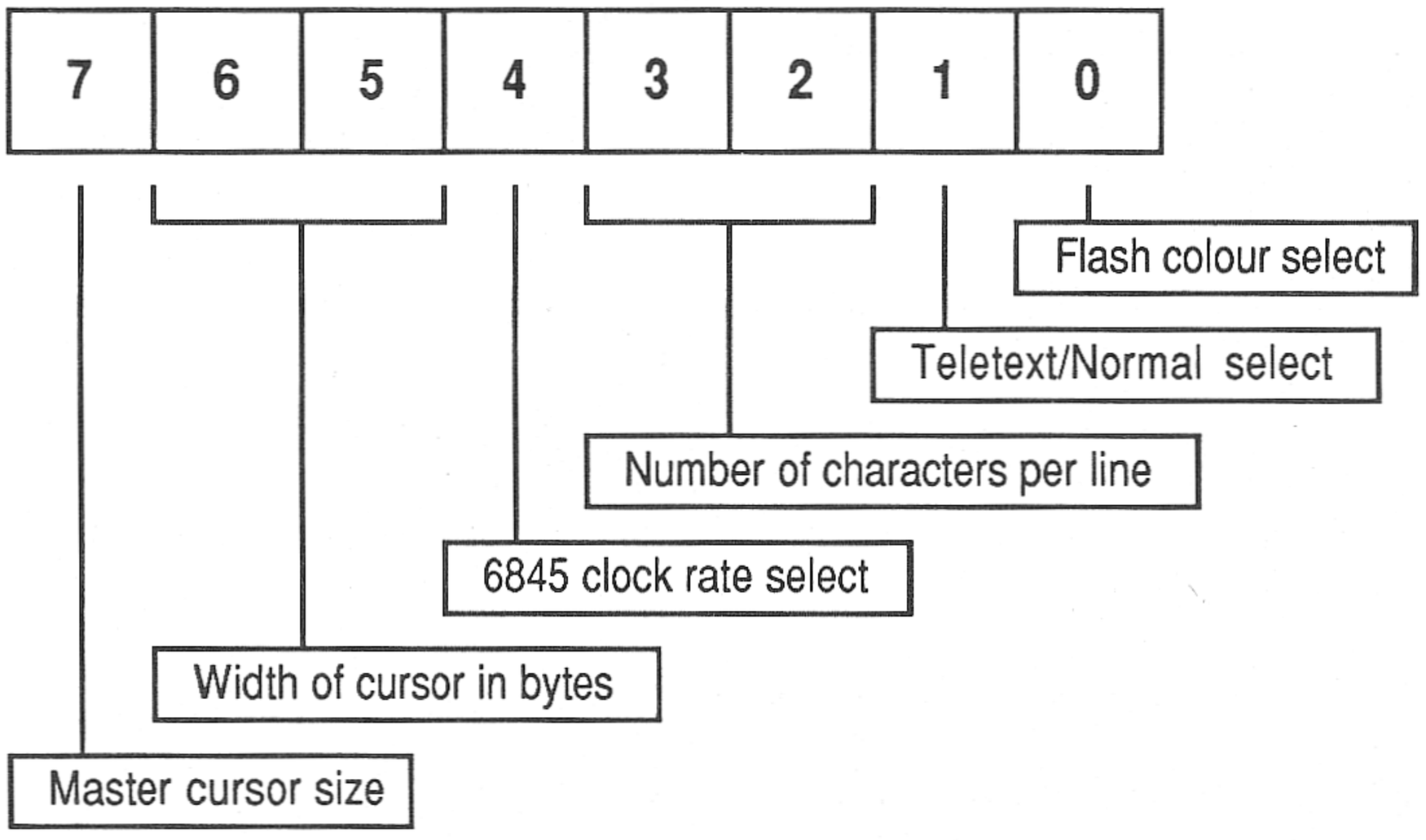


Fig 13.5 - The Video Control Register

Selected flash colour (bit 0)

This bit selects which colour of the two flashing colours is actually displayed at any particular time. It is continually changed by the operating system to generate the flashing colours. OSBYTE 9 and OSBYTE 10 control how long each colour is on the screen and can be defined down to one fiftieth of a second. By varying the flash rates of the colours it is possible to generate "new" colours. This is because a flash rate of one fiftieth of a second is fast enough to fool the eye into seeing a single colour rather than two rapidly flashing colours.

0 = first colour selected

1 = second colour selected

Teletext output select (bit 1)

This bit selects whether RGB input comes from the video serialiser in the ULA or from the teletext chip.

0 = on chip serialiser used

1 = teletext input selected

Number of characters per line (bits 2,3)

These two bits determine the actual number of displayed characters per line.

Bit 3	Bit 2	No. of chars. per line
1	1	80
1	0	40
0	1	20
0	0	10

6845 clock rate select (Bit 4)

The clock frequency sent to the 6845 can be varied using this bit.

0 = low frequency clock (modes 4-7)

1 = high frequency clock (modes 0-3)

Width of cursor (bits 5-7)

These three bits determine the cursor width as illustrated.

Cursor	Bit 7	Bit 6	Bit 5	Used for modes
-	1	0	0	0, 3, 4, 6
--	1	1	0	1, 5
----	1	1	1	2
-	0	1	0	7

To understand the operation of the cursor control, consider the relationship between the video data clock and each dot displayed on the screen. In two colour modes each dot is represented by one bit, in the four colour modes each dot is represented by two bits and in the 16 colour mode each dot is represented by four bits. Since each character is 8 dots wide, the video data clock must have 8, 16 or 32 cycles for each character width referred to the 6845. The cursor control bits 7-5 therefore enable the cursor for the first 8, the second 8 and the last 16 cycles of video data respectively.

For mode 7 the character is displayed in the character cell after it has been accessed by the 6845. This is due to the video data being generated from the special teletext chip rather than the video ULA serialiser. The cursor therefore has to be enabled in the second 8 cycles of the video data clock for each character.

NOTE - setting bits 5,6 and 7 to 0 will cause the cursor to vanish from the screen under ALL conditions.

General summary of the video control register

bit	7	6	5	4	3	2	1	0	value
Mode	cursor width			clock	chars.per line		Ttx.	flash	(hex.)
0	1	0	0	1	1	1	0	X	&9C
1	1	1	0	1	1	0	0	X	&D8
2	1	1	1	1	0	1	0	X	&F4
3	1	0	0	1	1	1	0	X	&9C
4	1	0	0	0	1	0	0	X	&88
5	1	1	0	0	0	1	0	X	&C4
6	1	0	0	0	1	0	0	X	&88
7	0	1	0	0	1	0	1	1	&4B

X signifies that the flash bit is changed regularly

13.3.14 The Palette - SHEILA &21 (write only)

The "Palette" is a 64 bit RAM in the video ULA which defines the relationship between the *logical* and *actual* colours displayed on the screen. If you don't understand the difference between *logical* and *actual* colours yet, then refer to the COLOUR section in the User Guide. *FX155 can be used to write colour data into the Palette. It will automatically EOR the physical colour with 7 (see later). Usually, it is better to use VDU19 or OSWORD &0C to program *logical* and *actual* colours.

The palette register consists of two 4 bit fields. Bits 0-3 are the *actual* colour field. Bits 4-7 are the *logical* colour field, as illustrated in figure 13.6.

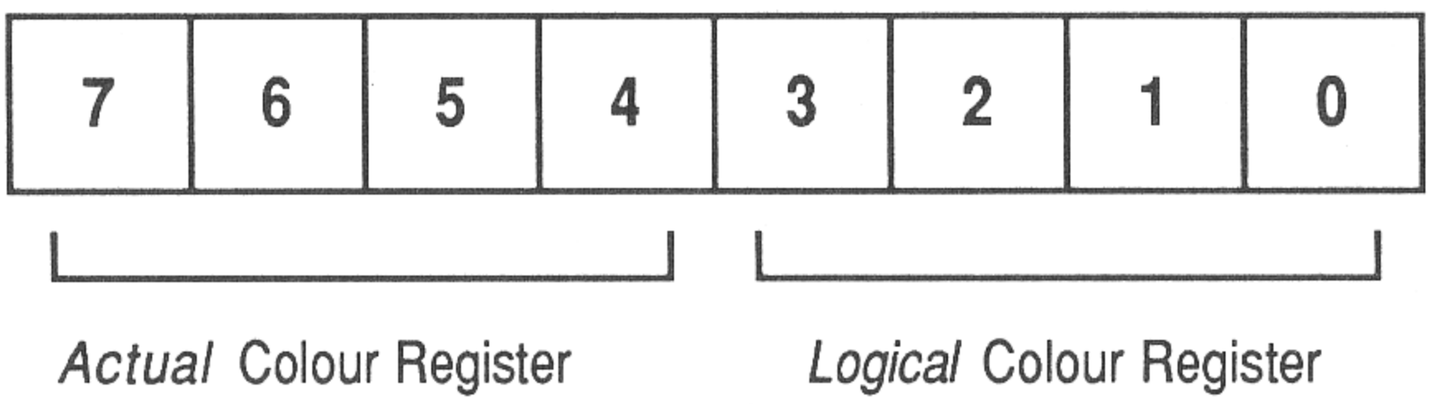


Fig 13.6 - The Palette register

Logical colour field

The following description of programming the palette only applies to direct programming using OSBYTE155. If OSWORD &0C or VDU19 are used, none of the problems which are about to be outlined will be relevant.

Programming the *logical* colour directly is easy in mode 2. The *logical* colour number then occupies the entire 4 bit field. In two colour modes 0,3,4 and 6, programming the *logical* colour directly is more complex. Bit 7 defines the *logical* colour, but bits 4,5 and 6 must be programmed to all their possible values. In other words, in order to set *logical* colour 1 to *actual* colour 5, it is necessary to program *logical* colours 8, 9, 10, 11, 12, 13, 14 and 15 to 5. If this is not done, some parts of characters will be in one colour and other parts will be in a different colour.

Programming the *logical* colours in a four colour mode is slightly more complex. Bits 7 and 5 together contain the *logical* colour number. All other possible combinations of bits 6 and 4 must also be programmed. The following table shows how to program *logical* colours 0-3. For example, to program *logical* colour 0, it is necessary to program four separate locations in the palette.

Logical Colour	palette reg. bit			
	7	6	5	4
0	0	0	0	0
	0	0	0	1
	0	1	0	0
	0	1	0	1
1	0	0	1	0
	0	0	1	1
	0	1	1	0
	0	1	1	1
2	1	0	0	0
	1	0	0	1
	1	1	0	0
	1	1	0	1
3	1	0	1	0
	1	0	1	1
	1	1	1	0
	1	1	1	1

General Summary for *logical* colour programming

Mode	Bit 7	Bit 6	Bit 5	Bit 4
2 colour	Logical colour Bit 0	x	x	x
4 colour	Logical colour Bit 1	x	Logical colour Bit 0	x
16 colour	Logical colour Bit 3	Logical colour Bit 2	Logical colour Bit 1	Logical colour Bit 0

Actual colour field

The *actual* colours are:

&00(0)	black
&01(1)	red
&02(2)	green
&03(3)	yellow (green-red)

&04(4)	blue
&05(5)	magenta (red-blue)
&06(6)	cyan (green-blue)
&07(7)	white
&08(8)	flashing black-white
&09(9)	flashing red-cyan
&0A(10)	flashing green-magenta
&0B(11)	flashing yellow-blue
&0C(12)	flashing blue-yellow
&0D(13)	flashing magenta-green
&0E(14)	flashing cyan-red
&0F(15)	flashing white-black

It is these colour numbers which should be used with OSBYTE 155. Note however that the actual number sent to the Palette is the above number EOR &07, i.e.. with the three colour bits inverted and the flash bit as above.

Some interesting effects using the palette

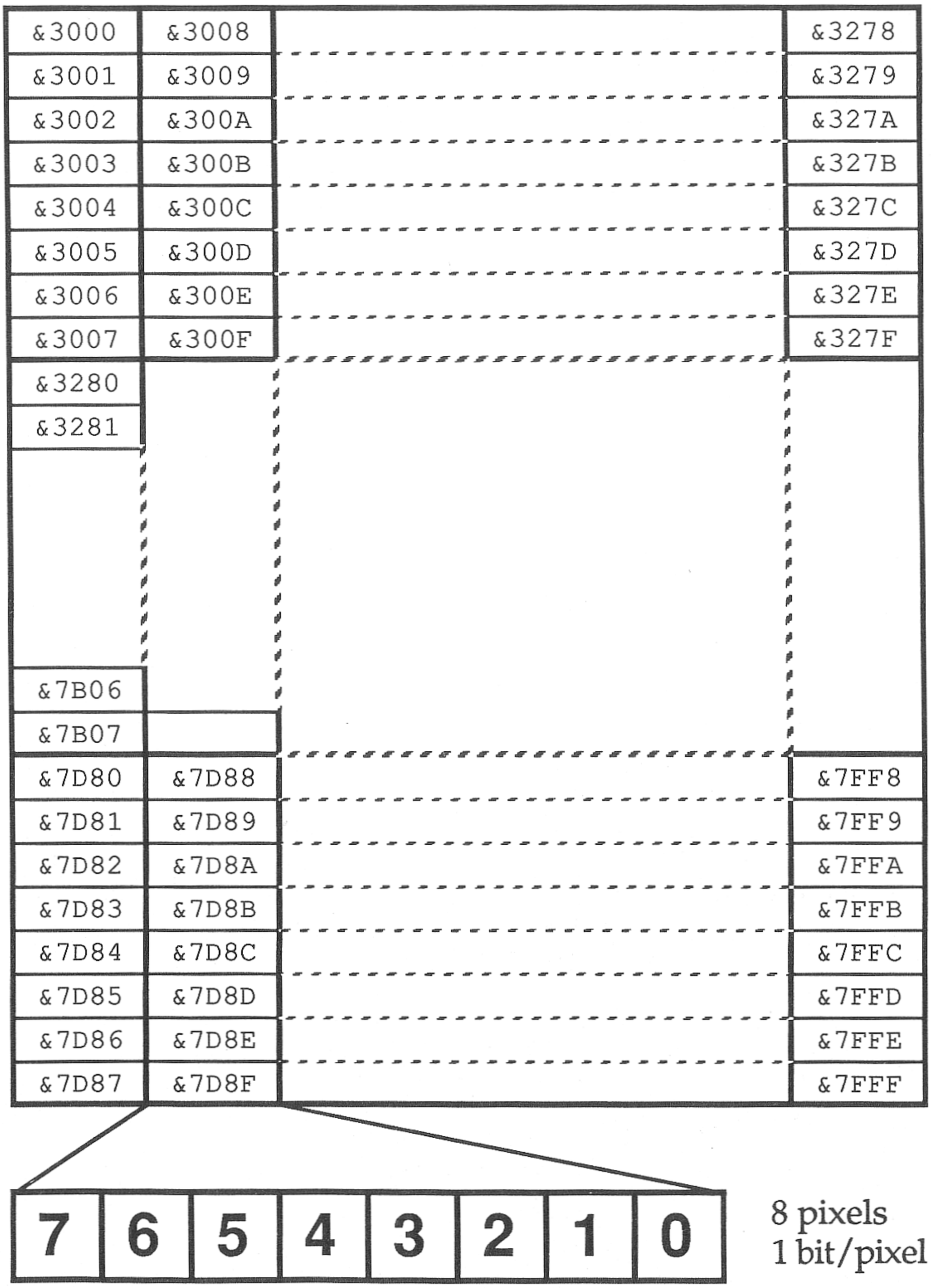
Because of the necessity to program 4 different palette locations for each colour in a four colour mode, some "nasty" effects can be produced on the screen if all four locations are not programmed with the same colour. To illustrate this point, try displaying four colours on the screen at once, then run this line of BASIC:

```
A%=155: REPEAT: X%=RND(255): CALL &FFF4: UNTIL0
```

13.4 Screen mode memory maps

MODE 0 Screen Layout

Graphics	640 x 256
Colours	2
Text	80 x 32

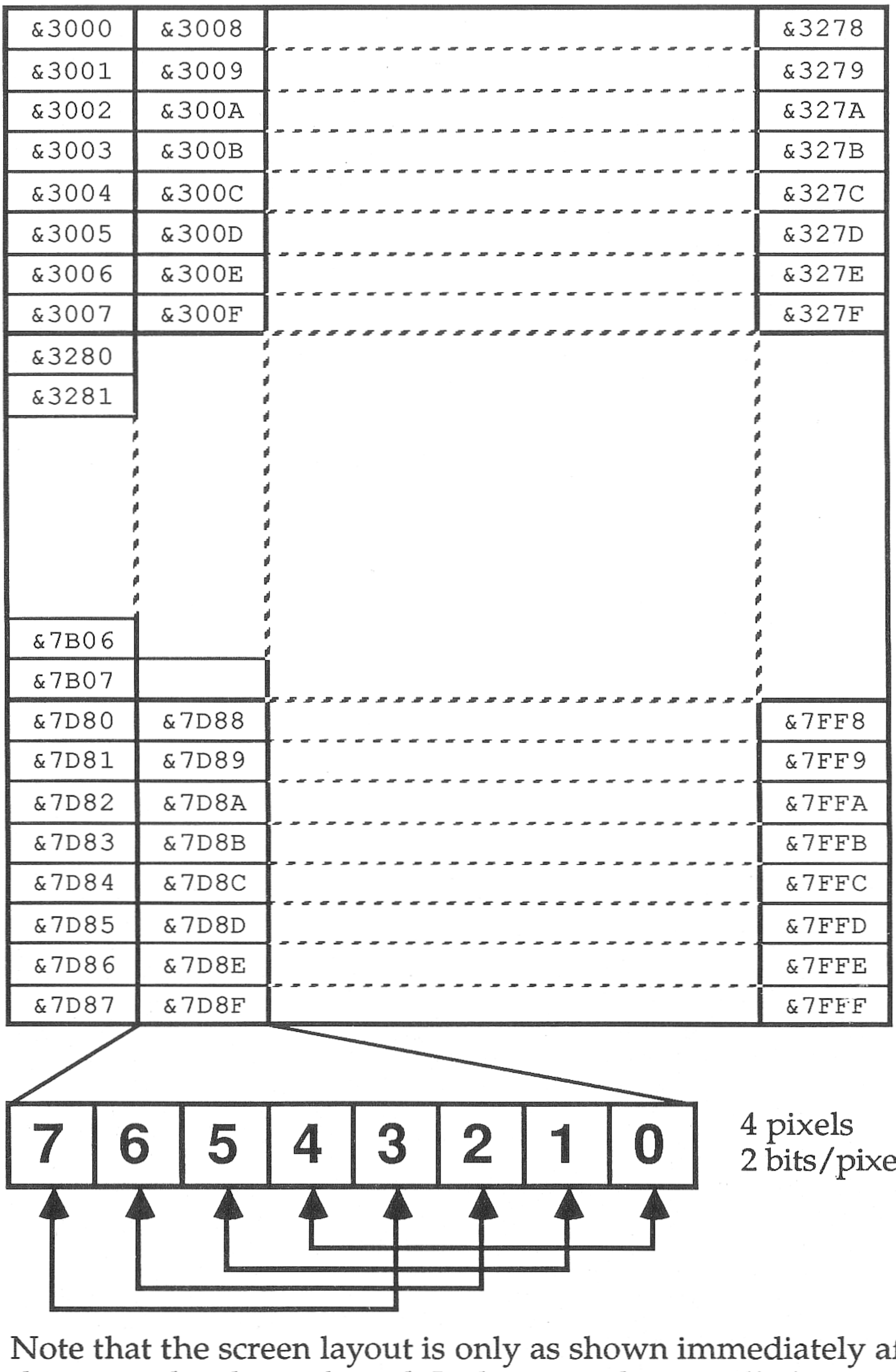


8 pixels
1 bit/pixel

Note that the screen layout is only as shown immediately after the screen has been cleared. It changes when scrolled.

MODE 1 Screen Layout

Graphics 320 x 256
Colours 4
Text 40x 32

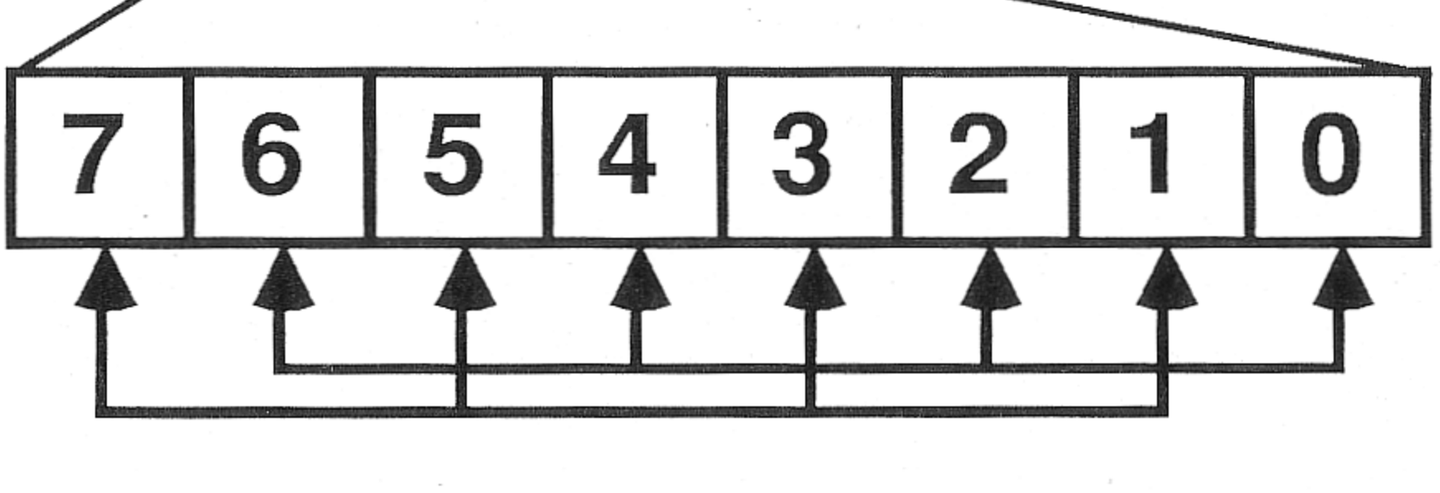


Note that the screen layout is only as shown immediately after the screen has been cleared. It changes when scrolled.

MODE 2 Screen Layout

Graphics 160x 256
Colours 8
Text 20x 32

&3000	&3008		&3278
&3001	&3009		&3279
&3002	&300A		&327A
&3003	&300B		&327B
&3004	&300C		&327C
&3005	&300D		&327D
&3006	&300E		&327E
&3007	&300F		&327F
&3280			
&3281			
&7B06			
&7B07			
&7D80	&7D88		&7FF8
&7D81	&7D89		&7FF9
&7D82	&7D8A		&7FFA
&7D83	&7D8B		&7FFB
&7D84	&7D8C		&7FFC
&7D85	&7D8D		&7FFD
&7D86	&7D8E		&7FFE
&7D87	&7D8F		&7FFF



2pixels
4 bits/pixel

Note that the screen layout is only as shown immediately after the screen has been cleared. It changes when scrolled.

MODE 3 Screen Layout

Graphics Not available
Colours 2
Text 80 x 25

&4000	&4008		&4278
&4001	&4009		&4279
&4002	&400A		&427A
&4003	&400B		&427B
&4004	&400C		&427C
&4005	&400D		&427D
&4006	&400E		&427E
&4007	&400F		&427F
BLANK	BLANK		BLANK
BLANK	BLANK		BLANK
&4280			
&7987			
BLANK	BLANK		BLANK
BLANK	BLANK		BLANK
&7C00	&7C08		&7E78
&7C01	&7C09		&7E79
&7C02	&7C0A		&7E7A
&7C03	&7C0B		&7E7B
&7C04	&7C0C		&7E7C
&7C05	&7C0D		&7E7D
&7C06	&7C0E		&7E7E
&7C07	&7C0F		&7E7F
BLANK	BLANK		BLANK
BLANK	BLANK		BLANK

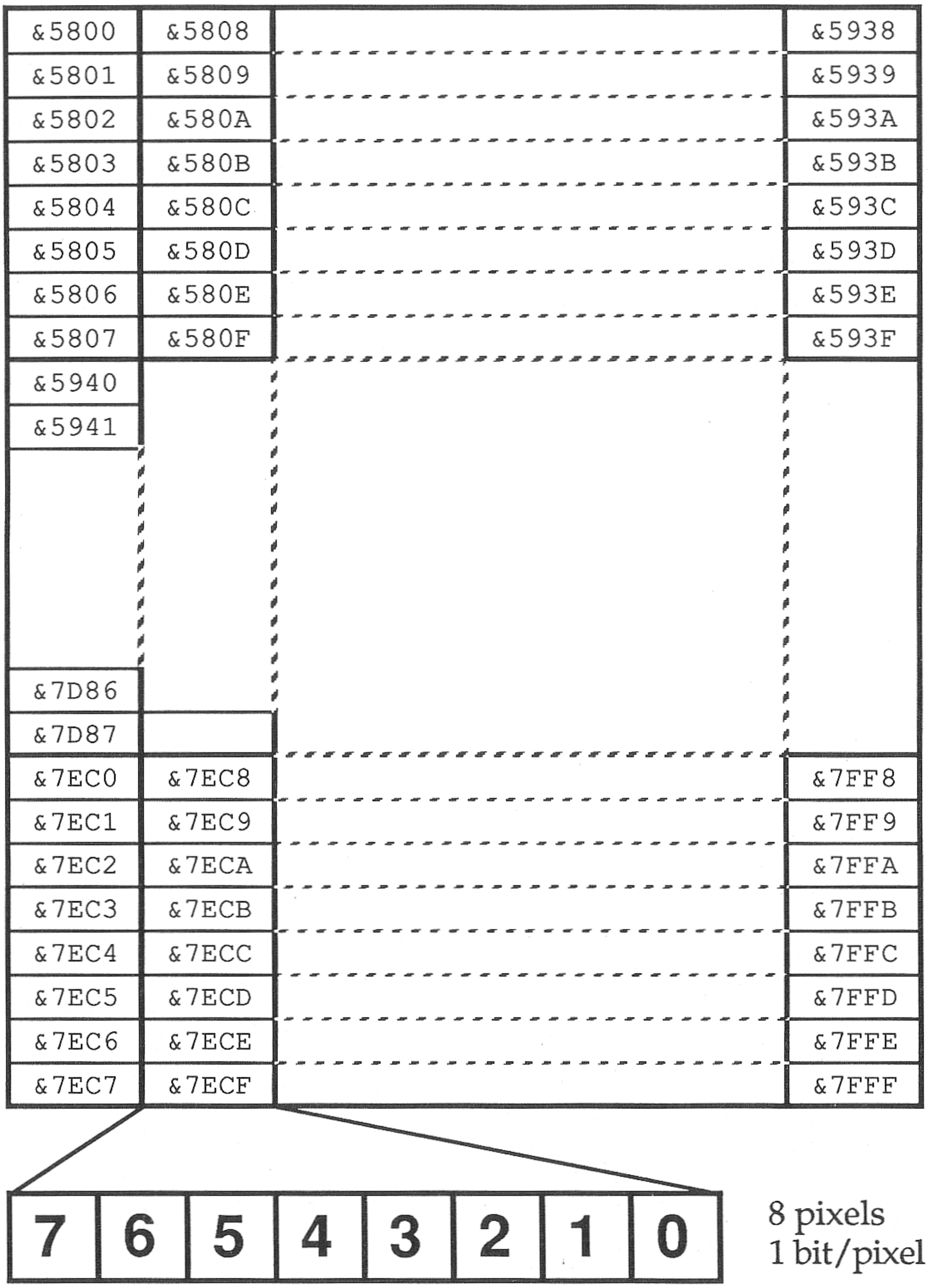
7	6	5	4	3	2	1	0

8 pixels
1 bit/pixel

Note that the screen layout is only as shown immediately after the screen has been cleared. It changes when scrolled.

MODE 4 Screen Layout

Graphics 320 x 256
Colours 2
Text 40x 32



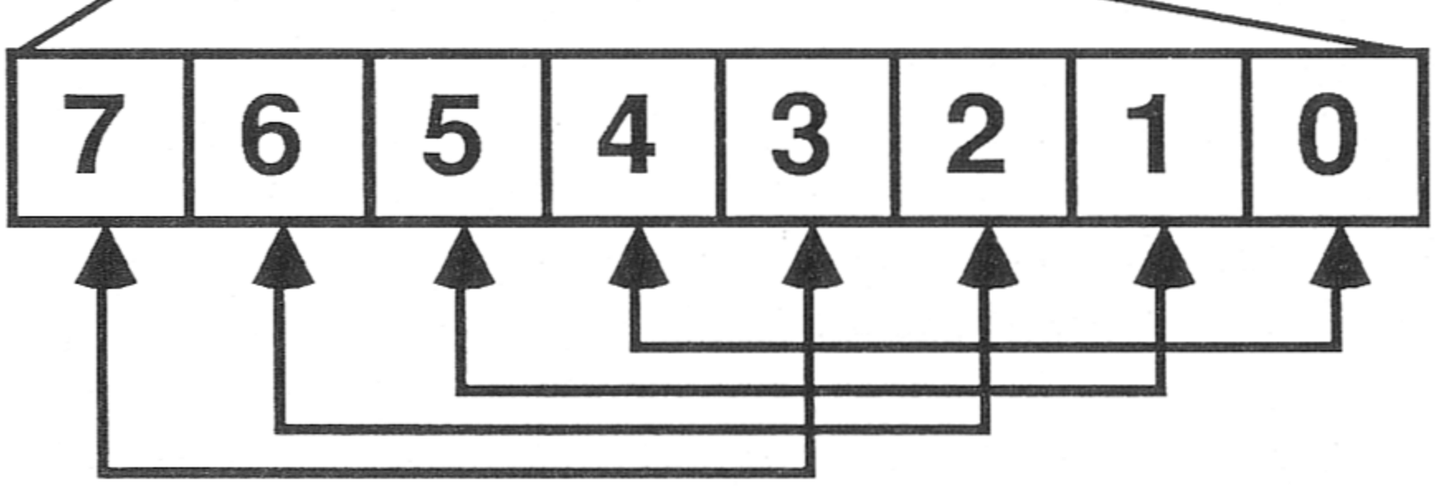
8 pixels
1 bit/pixel

Note that the screen layout is only as shown immediately after the screen has been cleared. It changes when scrolled.

MODE 5 Screen Layout

Graphics 160 x 256
Colours 4
Text 20x 32

&5800	&5808		&5938
&5801	&5809		&5939
&5802	&580A		&593A
&5803	&580B		&593B
&5804	&580C		&593C
&5805	&580D		&593D
&5806	&580E		&593E
&5807	&580F		&593F
&5940			
&5941			
&7D86			
&7D87			
&7EC0	&7EC8		&7FF8
&7EC1	&7EC9		&7FF9
&7EC2	&7ECA		&7FFA
&7EC3	&7ECB		&7FFB
&7EC4	&7ECC		&7FFC
&7EC5	&7ECD		&7FFD
&7EC6	&7ECE		&7FFE
&7EC7	&7ECF		&7FFF



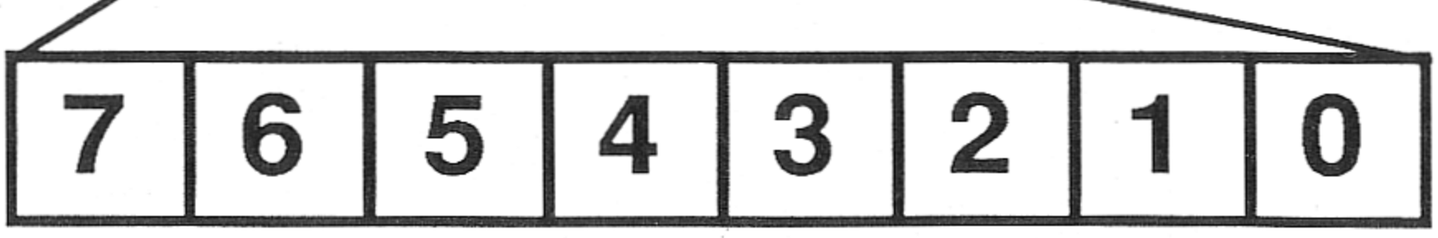
4 pixels
2 bits/pixel

Note that the screen layout is only as shown immediately after the screen has been cleared. It changes when scrolled.

MODE 6 Screen Layout

Graphics Not available
Colours 2
Text 40 x 25

&6000	&6008		&6138
&6001	&6009		&6139
&6002	&600A		&613A
&6003	&600B		&613B
&6004	&600C		&613C
&6005	&600D		&613D
&6006	&600E		&613E
&6007	&600F		&613F
BLANK	BLANK		BLANK
BLANK	BLANK		BLANK
&6140			
&7CC7			
BLANK	BLANK		BLANK
BLANK	BLANK		BLANK
&7E00	&7E08		&7F38
&7E01	&7E09		&7F39
&7E02	&7E0A		&7F3A
&7E03	&7E0B		&7F3B
&7E04	&7E0C		&7F3C
&7E05	&7E0D		&7F3D
&7E06	&7E0E		&7F3E
&7E07	&7E0F		&7F3F
BLANK	BLANK		BLANK
BLANK	BLANK		BLANK

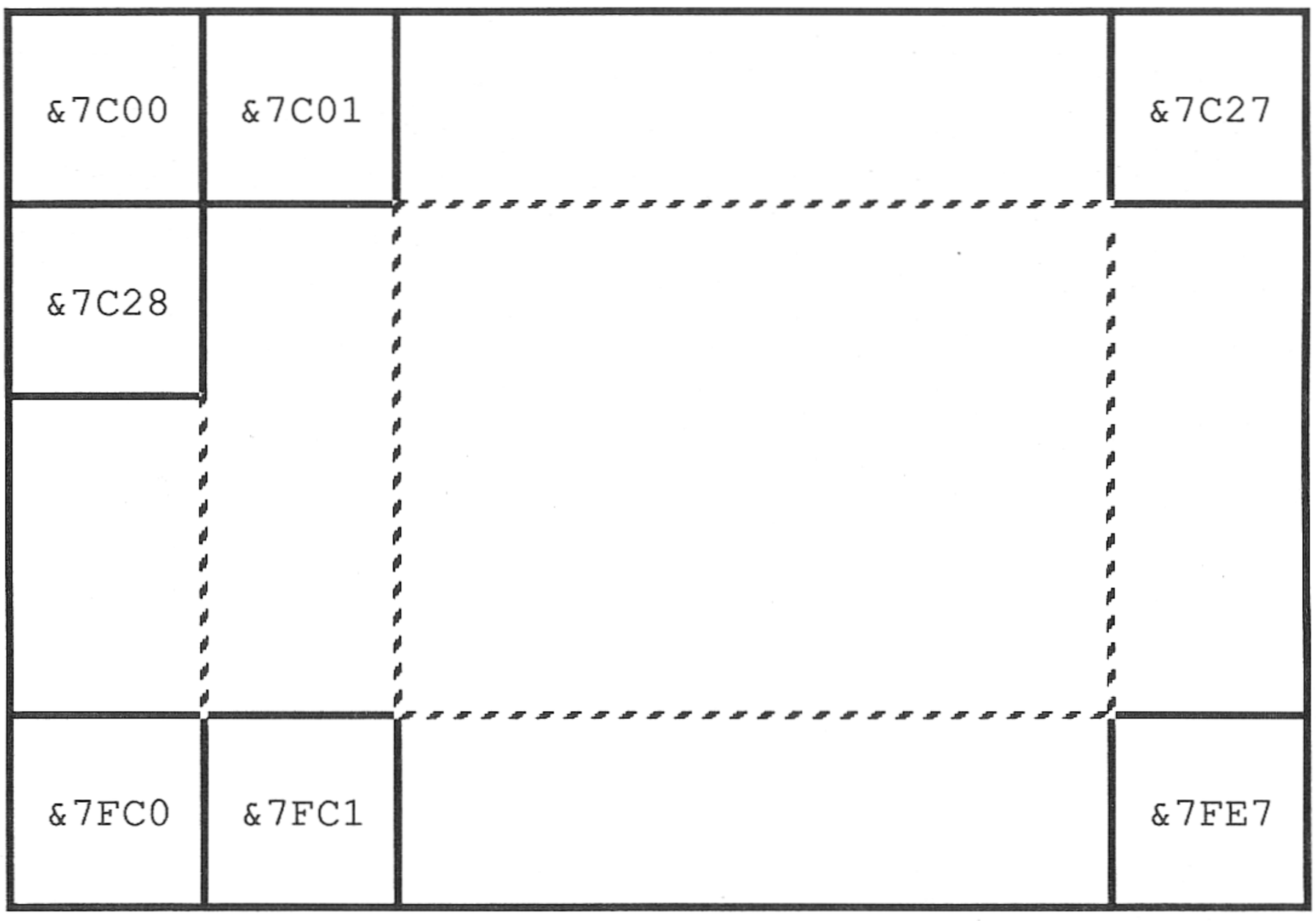


8 pixels
1 bit/pixel

Note that the screen layout is only as shown immediately after the screen has been cleared. It changes when scrolled.

MODE 7 Screen Layout

Graphics	Teletext graphics only
Colours	Teletext
Text	40x 25



Note that the screen layout is only as shown immediately after the screen has been cleared. It changes when scrolled.

14 Keyboard routines

This chapter describes the routines which control the keyboard. The Master series computers, BBC model B and the Electron have different keyboard matrix layouts and this should be kept in mind when producing software which is intended to be compatible on all machines.

On the Master Compact additional use is made of the cursor keys as part of the analogue joystick emulation. Further information about the Compact joystick arrangements are described in chapter 20.

The BREAK key is a direct link to the 6502 reset line and does not appear on the keyboard matrix of any of the Acorn BBC series machines.

The keyboard on the Master series computers consists of a total of 93 keys; 74 in the main keyboard section and 19 in the numeric keypad. The Master Series computers have a different matrix arrangement from model Bs to allow for the additional keys. The start-up option switch found on the keyboard circuit board of the BBC model B micros has been replaced by non-volatile memory on Master series computers.

The keyboard software in the operating system is interrupt driven. The system 6522 constantly scans the keyboard matrix columns. When a key is pressed an interrupt is generated and the scanning action of the 6522 is stopped. The keyboard reads the column counter to identify which column of the keyboard has been selected. Each keyboard matrix row is then examined to precisely locate the key. The keyboard software operates a '2 key rollover', which means that a second key press rapidly after the first key press will also be recognised even if the two key presses overlap.

Having scanned the keyboard matrix the *internal key number* of that key is stored in the *current keys pressed information* and a flag is set to indicate that a key has been pressed. At this stage the interrupt routine returns to the foreground task.

The conversion of the internal key number to an ASCII code, with adjustments according to the state of the SHIFT, CTRL keys etc., takes place during the general background processing routine, which the operating system checks every 10 ms. The operation is complete when the final character value has been placed in the keyboard buffer.

14.1 Key values

The table below gives the values used by the operating system to distinguish the different keys. The internal key numbers represent the lowest level of translation from the matrix addresses. These numbers exclusive-ORed with &FF give the INKEY negative numbers which are translated via a table to give the ASCII codes. Different operating system routines use different key-value types. The ASCII values are returned by OSRDCH while OSBYTE &78 uses the internal key number (IK No. in the table). The INKEY numbers describe values used by the BASIC INKEY instruction when performing keyboard scans.

The Electron key values are identical to the BBC model B but the option switches have not been implemented.

Character	ASCII		INKEY		IK No.	
	dec.	hex.	dec.	hex	dec	hex
SPACE	32	20	- 99	9D	98	62
!	33	21				
"	34	22				
#	35	23				
$	36	24				
%	37	25				
&	38	26				
'	39	27				
(	40	28				
)	41	29				
*	42	2A				
+	43	2B				
,	44	2C	-103	99	102	66
-	45	2D	- 24	E8	23	17
.	46	2E	-104	98	103	67
/	47	2F	-121	87	104	68
0	48	30	- 40	D8	39	27
1	49	31	- 49	CF	48	30
2	50	32	- 50	CE	49	31
3	51	33	- 18	EE	17	11
4	52	34	- 19	ED	18	12
5	53	35	- 20	EC	19	13
6	54	36	- 53	CB	52	34
7	55	37	- 37	DB	36	24
8	56	38	- 22	EA	21	15
9	57	39	- 39	D9	37	25
:	58	3A	- 73	B7	72	48
;	59	3B	- 88	A8	87	57
<	60	3C				
=	61	3D				
>	62	3E				
?	63	3F				

Character	ASCII		INKEY		IK No.	
	dec.	hex.	dec.	hex	dec	hex
@	64	40	- 72	B8	71	47
A	65	41	- 66	BE	65	41
B	66	42	-101	9B	100	64
C	67	43	- 83	AD	82	52
D	68	44	- 51	CD	50	32
E	69	45	- 35	DD	34	22
F	70	46	- 68	BC	67	43
G	71	47	- 84	AC	83	53
H	72	48	- 85	AB	84	54
I	73	49	- 38	DA	38	26
J	74	4A	- 70	BA	69	45
K	75	4B	- 71	B9	70	46
L	76	4C	- 87	A9	86	56
M	77	4D	-102	9A	101	65
N	78	4E	- 86	AA	85	55
O	79	4F	- 55	C9	54	36
P	80	50	- 56	C8	55	37
Q	81	51	- 17	EF	16	10
R	82	52	- 52	CC	51	33
S	83	53	- 82	AE	81	51
T	84	54	- 36	DC	35	23
U	85	55	- 54	CA	53	35
V	86	56	-100	9C	99	63
W	87	57	- 34	DE	33	21
X	88	58	- 67	BD	66	42
Y	89	59	- 69	BB	68	44
Z	90	5A	- 98	9E	97	61
[	91	5B	- 57	C7	56	38
\	92	5C	-121	87	120	78
]	93	5D	- 89	A7	88	58
^	94	5E	- 25	E7	24	18
_	95	5F	- 41	D7	40	28

Character	ASCII		INKEY		IK No.	
	dec.	hex.	dec.	hex	dec	hex
`	96	60				
a	97	61				
b	98	62				
c	99	63				
d	100	64				
e	101	65				
f	102	66				
g	103	67				
h	104	68				
i	105	69				
j	106	6A				
k	107	6B				
l	108	6C				
m	109	6D				
n	110	6E				
o	111	6F				
p	112	70				
q	113	71				
r	114	72				
s	115	73				
t	116	74				
u	117	75				
v	118	76				
w	119	77				
x	120	78				
y	121	79				
z	122	7A				
{	123	7B				
|	124	7C				
}	125	7D				
~	126	7E				

Character	ASCII		INKEY		IK No.	
	dec.	hex.	dec.	hex	dec	hex
Escape	27	1B	-113	8F	112	70
Tab	9	9	- 97	9F	86	60
Capslk			- 65	BF	64	40
Ctrl			- 2	FE	1	1
Sh. Lck			- 81	AF	80	50
Shift			- 1	FF	0	0
Delete	127	7F	- 90	A6	89	59
Copy	135	87	-106	96	105	69
Return	13	D	- 74	B6	73	49
Up Curs	139	8B	- 58	C6	57	39
Dn Curs	138	8A	- 42	D6	41	29
L. Curs	136	88	- 26	E6	25	19
R. Curs	137	89	-122	86	121	79
f0			- 33	DF	32	20
f1			-114	8E	113	71
f2			-115	8D	114	72
f3			-116	8F	115	73
f4			- 21	EB	20	14
f5			-117	8B	116	74
f6			-118	8C	117	75
f7			- 23	E9	22	16
f8			-119	8D	118	76
f9			-120	8E	119	77

Master Series Keypad

Character	ASCII		INKEY		IK No.	
	dec.	hex.	dec.	hex	dec	hex
0	48	30	-107	95	106	6A
1	49	31	-108	94	107	6B
2	50	32	-125	83	124	7C
3	51	33	-109	93	108	6C
4	52	34	-123	85	122	7A
5	53	35	-124	84	123	7B
6	54	36	-27	E5	26	1A
7	55	37	-28	E4	27	1B
8	56	38	-43	D5	42	2A
9	57	39	-44	D4	43	2B
+	43	2B	-59	C5	58	3A
-	45	2D	-60	C4	59	3B
/	47	2F	-75	B5	74	4A
£	35	23	-91	A5	90	5A
*	42	2A	-92	A4	91	5B
,	44	2C	-93	A3	92	5C
Return	13	0D	-61	C3	60	3C
Delete	127	7F	-76	B4	75	4B

Start up option switch (model B)

Character	ASCII		INKEY		IK No.	
	dec.	hex.	dec.	hex	dec	hex
bit 0					9	9
bit 1					8	8
bit 2					7	7
bit 3					6	6
bit 4					5	5
bit 5					4	4
bit 6					3	3
bit 7					2	2

14.2 Read key with time limit OSBYTE call

Call address &FFF4
Indirected through &20A
A=&81 (129)

This call is functionally equivalent to BASIC's INKEY command and can be used get a character from the keyboard with a time limit, scan the keyboard for a particular key press or return information about the OS type (described in section 24.4.1).

(a) Read key with time limit

Entry parameters:
X and Y specify time limit in centiseconds
(low byte, high byte)

Maximum time limit is &7FFF centiseconds (5.5 minutes approx.)

On exit:
If a character is detected, X=ASCII value of key,
Y=0 and C=0
If a character is not detected by timeout then Y=&FF
and C=1
If Escape is pressed then Y=&1B (27) and C=1

(b) Scan keyboard for key press

Entry parameters:
X=negative INKEY value for key to be scanned
Y=&FF

On exit:
X and Y contain &FF if the key being scanned is pressed.

14.3 Keyboard scan OSBYTE call

Call address &FFF4
Indirected through &20A
A=&79 (121)

It should be noted that on the BBC model B the key matrix is not scanned in a regular ascending order. However, on other machines in the Acorn BBC range the keyboard scan is performed in ascending numerical order. This call returns information about the first pressed key encountered during the scan. Other keys may have been pressed as well,

and a further call or calls will be needed to complete the entire keyboard scan. The BBC microcomputer keys are scanned in the order :

&10, &20 ... &70, &11, &21 ... &71 etc. to &19, &29 ... &79

Entry parameters:
X determines the key to be detected and also
determines the range of keys to be scanned.

Key numbers refer to internal key numbers given in the table in section 14.1.

(a) To scan a particular key:
X=key number EOR &80
on exit X<0 if the key is pressed

(b) To scan the matrix starting from a particular key number:
X=key number

After call X=key number of any key pressed or &FF if no key pressed.

On exit:
A is preserved
X contains key value (see above)
Y and C are undefined

14.4 Keyboard scan from &10 OSBYTE call

Call address &FFF4
Indirected through &20A
A=&7A (122)

Internal key number (see table above) of the key pressed is returned in X.

This call is directly equivalent to an OSBYTE call with A=&79 and X=16.

On exit:
A is preserved
X contains key number or zero if none pressed
Y and C are undefined

14.5 Write current keys pressed OSBYTE call

Call address &FFF4
Indirected through &20A
A=&78 (120)

This call should only be made by filing systems which have recognised a key pressed with BREAK and are initialising accordingly (see paged ROM service call with A=&03, section 17.4.1). This call should be used to write the old key pressed value to prevent its entry into the keyboard buffer.

The operating system operates a two key roll-over for keyboard input (recognising a second key press even when the first key is still pressed). There are two zero page locations which contain the values of the two key-presses which may be recognised at any one time.

Entry parameters : X and Y contain values to be written

Value in X is stored as the old key information
Value in Y is stored in the new key information

On exit:

A, X and Y are preserved
C is undefined

14.6 Read key translation table address OSBYTE call

Call address &FFF4
Indirected through &20A
A=&AC (172) - address low byte
A=&AD (173) - address high byte

This call is implemented on all computers in the Acorn BBC range. However, it should be noted that the table is hardware specific due to the different keyboard matrix layouts on different machines. When used across the Tube the address returned refers to the i/o processor's memory.

Use of this call is not recommended.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

On exit:

X=low byte
Y=high byte

14.7 Set keyboard auto-repeat delay OSBYTE call

Call address &FFF4
Indirected through &20A
A=&0B (11)

Entry parameters:
X determines delay before repeating starts, Y=0

X=0 Disables auto-repeat facility
X=n Sets delay to n centiseconds
(n=32 is the default setting)

On exit:
A is preserved
X contains the old delay setting
Y and C are undefined

Read/write keyboard auto-repeat delay OSBYTE call

A=&C4 (196)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old delay value is returned in X. The contents of the next location are returned in Y.

14.8 Set keyboard auto-repeat period

Call address &FFF4
Indirected through &20A
A=&0C (12)

Entry parameters:
X determines auto-repeat periodic interval, Y=0

X=0 Resets delay and repeat to default vals
X=n Sets repeat interval to n centiseconds
(n=8 is the default value)

On exit:
A is preserved
X contains the old *FX 12 setting
Y and C are undefined

Read/write keyboard auto-repeat period

A=&C5 (197)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old period value is returned in X. The contents of the next location are returned in Y.

14.9 Function keys and user definable keys

The function keys and certain other keys may have user defined strings associated with them using the '*KEY' command.

The following calls have various effects on these keys.

14.9.1 Enable/disable cursor editing OSBYTE call

Call address &FFF4
Indirected through &20A
A=&04 (4)

Entry parameters:
X determines editing keys' status, Y=0

X=0 Enable cursor editing (default setting)
X=1 Disable cursor editing
edit keys give ASCII codes
X=2 Disable cursor editing
edit keys act as soft keys (11 to 15)
X=3 On the Master Compact only
makes the cursor keys have joystick like effect
and the copy key simulates the fire button.

On exit:
A is preserved
X contains the previous status of the editing keys
Y and C are undefined

Read/write cursor editing status OSBYTE call

A=&ED (237)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old status value is returned in X.

14.9.2 Read/write function key status OSBYTE calls

Call address &FFF4
Indirected through &20A
A=&E1 (225) - function keys, &80 to &8F
A=&E2 (226) - SHIFT+function keys, &90 to &9F
A=&E3 (227) - CTRL+function keys, &A0 to &AF
A=&E4 (228) - CTRL+SHIFT+fn keys, &B0 to &BF

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old status is returned in X. The contents of the next location are returned in Y.

These locations determine the action taken by the operating system when a function key is pressed.

value 0	totally ignore key.
value 1	expand as normal soft key.
value 2 to &FF	add n (base) to soft key number to provide "ASCII" code.

The default settings are :-

fn keys alone	&01	expand using soft key strings
fn keys+SHIFT	&80	code &80+soft key number
fn keys+CTRL	&90	code &90+soft key number
fn keys+SHIFT+CTRL	&00	key has no effect

14.9.3 Read/write character status flag OSBYTE calls

Call address &FFF4
Indirected through &20A
A=&DD (221) - characters &C0 to &CF
A=&DE (222) - characters &D0 to &DF
A=&DF (223) - characters &E0 to &EF
A=&E0 (224) - characters &F0 to &FF

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old status value is returned in X. The contents of the next location are returned in Y.

These locations determine the effect of the character values &C0 (192) to &FF (255) when placed in the input buffer. The meanings of these flag values are the same as for those calls described in the section above.

Default values are &01,&D0,&E0 and &F0 (respectively).

14.9.4 Read/write length of soft key string OSBYTE call

Call address &FFF4
Indirected through &20A
A=&D8 (216)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old length is returned in X.

This location contains the number of characters remaining in the soft key buffer of the current soft key expansion.

14.9.5 Read/write soft key consistency flag OSBYTE call

Call address &FFF4
Indirected through &20A
A=&F4 (244)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag is returned in X.

A non zero value indicates that the soft key buffer is in an inconsistent state (the operating system does this during soft key string entries and deletions). If the soft keys are in an inconsistent state during a soft break then the soft key buffer is cleared (otherwise it is preserved).

14.9.6 Reset function keys OSBYTE call

Call address &FFF4
Indirected through &20A
A=&12 (18)

Function key definitions are normally preserved other than following a hard break. This call allows the set of definitions to be cleared.

No entry parameters.

On exit:
X is undefined
Y is preserved.

14.10 Reflect keyboard status in keyboard LEDs OSBYTE

Call address &FFF4
Indirected through &20A
A=&76 (118)

The keyboard status byte maintains a record of the current setting of caps-lock and shift-lock and is used by the keyboard interrupt routine which also updates the keyboard caps-lock and shift-lock LEDs. When this byte is written directly using OSBYTE &CA this call can be used to ensure that the keyboard LEDs reflect the current keyboard status byte.

On exit,
A is preserved
X has bit 7 set if Ctrl is pressed
Y is undefined

14.11 Read/write keyboard disable OSBYTE call

Call address &FFF4
Indirected through &20A
A=&C9 (201)

This location is used on the all Acorn BBC machines. However this call should only be made by the Econet filing system.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

If location contains 0 then scan keyboard normally, otherwise ignore all keys except BREAK.

14.12 Read/write keyboard status byte OSBYTE call

Call address &FFF4
Indirected through &20A
A=&CA (202)

This location is used in a similar manner on all the Acorn BBC machines. The value of the status byte has a different meaning on the Electron compared to other machines in the range, reflecting its different keyboard design.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old status byte value is returned in X.

(a) The BBC microcomputer

bit 3 - 1 if SHIFT is pressed.
bit 4 - 0 if CAPS LOCK is engaged.
bit 5 - 0 if SHIFT LOCK is engaged.
bit 6 - 1 if CTRL is pressed.
bit 7 - 1 SHIFT enabled, if a LOCK key is engaged then SHIFT reverses the LOCK.

SHIFT enable (bit 7) may be set by holding SHIFT down as the CAPS LOCK key is engaged which enables lower-case letters to be typed when capitals are selected by pressing the required key plus SHIFT. The only way to set SHIFT enable for the SHIFT LOCK key is to use *FX202,144 (or OSBYTE &CA).

(b) The Electron

bit 4 - 0 if CAPS LOCK active
bit 5 - 1 if Fn active
bit 6 - 1 if SHIFT active
bit 7 - 1 if CTRL active

All bits except the CAPS LOCK bit will only change transiently and are subsequently unlikely to be of use.

14.13 Read/write keyboard semaphore OSBYTE call

Call address &FFF4
Indirected through &20A
A=&B2 (178)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag is returned in X.

If this location contains 0 then keyboard interrupts are ignored. Keyboard interrupts are enabled if it contains &FF.

14.14 Set base for numeric keypad OSBYTE call

Call address &FFF4
Indirected through &20A
A=&EE (238)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag is returned in X.

This call is only available in Master series computers. The ASCII characters generated by the keys in the keypad are determined by the value of this location. The value generated is calculated as the ASCII value of the keytop legend minus 48 plus this value.

14.15 Read/write shift key effect OSBYTE call

Call address &FFF4
Indirected through &20A
A=&FE (254)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag is returned in X.

On the Master series this call may be used to enable the use of the shift key with the numeric keypad. When numeric shift is enabled the character generated by the shifted key will be the alternative character generated by the equivalent main keyboard key.

14.16 Electron firm keys

The Electron operating system has a mechanism whereby the language ROM is able to insert strings of text into the input buffer in response to certain function key combinations. For more information about firm keys see Language ROMs section 17.3.2.

14.16.1 Read/write firm key pointer OSBYTE call

Call address &FFF4
Indirected through &20A
A=&CC (204)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old pointer value is returned in X. The value in the next location is returned in Y.

The value contained in this location is a pointer into the currently expanding firm key.

14.16.2 Read/write length of firm key string OSBYTE call

Call address &FFF4
Indirected through &20A
A=&CD (205)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old length value is returned in X.

This location contains the length of the string currently being expanded from a firm key.

14.17 Read/write TAB key character OSBYTE call

Call address &FFF4
Indirected through &20A
A=&DB (219)

This call has a different function on the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old character value is returned in X.

This location contains the value to be returned by the TAB key. It is possible to use the TAB key as a soft key by setting this location to &80+n where n is the soft key number. If the key value selected normally responds to CTRL and SHIFT combinations the TAB key will respond accordingly.

Default value is 9 (forward cursor 1 character space).

14.18 Read/write Escape character OSBYTE call

Call address &FFF4
Indirected through &20A
A=&DC (220)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old character value is returned in X.

This location contains the ASCII character (and key) which will generate an ESCAPE condition or event. The default value is &1B(27)

15 Serial I/O (RS232/423)

A considerable amount of jargon and confusion surrounds the subject of serial communications. Serial technology has been around for 150 years or more and much of the terminology dates from the first intercontinental serial transmission in 1866. The RS232 standard (a much misused and abused term) dates from a proposal in 1959. The formal standard is embodied in Recommended Standard Number 232, Revision C from the Engineering Department of the Electronic Industries Association (EIA RS232C) better known to most of us as RS232. This document, written in the heavy stilted prose of the professional engineer and laced with obscure and esoteric jargon, has been responsible for much of the confusion about this 'standard'. The basic need for a standard method for the transmission of data between computer devices has held the standard loosely together. As is often the case in the computer industry the establishment of a market leading product has greatly contributed to the standard. In the same way that the Centronics printer interface became the parallel printer standard,Hayes have influenced the serial I/O interface standard with their modems.

There are a multitude of different sub-sets and super-sets of the RS232 standard and it appears that few manufacturers have the courage not to include the magic characters RS232 in their publicity material. The truth is that while complying to parts of the RS232 standard may impart the legal right to call an interface RS232, in practice it can be an extremely frustrating ordeal trying to persuade two such RS232 designated devices to talk to each other.

The Acorn philosophy has been to trim the hardware link to the bearest minimum by using a minimum number of control lines. This does at least reduce the number of decisions about which wire to solder to which pin. The adoption of the RS423 standard in the first BBC Micro and the subsequent return to RS232 in the Master Series may be ignored as the serial hardware between the two machines seems largely unchanged. The RS423 standard is intended to describe an RS232 compatible interface capable of use over greater distances.

15.1 The RS232C standard

This section does not propose to be an exhaustive description of the RS232C standard but it is hoped that the information will provide a greater understanding of the standard in the context of devices other than the Acorn BBC series of microcomputers.

The formal name of the standard is "Interface between *Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interchange*." The terms Data Terminal Equipment (DTE) and Data Communication Equipment (DCE) are, in a microcomputing context, normally a computer and a modem respectively. The DCE may also be another computer, printer or some other peripheral with a serial interface. The terms DTE and DCE will be used in the following description to avoid possible ambiguities.

The EIA RS232 document covers three different aspects of the serial communications interface:

1. Mechanical description of interface circuits
2. Functional description of interchange circuits
3. Electrical signal characteristics

15.1.1 Mechanical description of interface circuits

The RS232 standard does not specify the D-25 connector commonly used for RS232, it says only that there should be two connectors, female for the DCE and male for the DTE. The IBM PC has a male connector, of the D-25 type (although the IBM PC/AT uses a D-9 connector).

A total of 22 interchange circuits are defined and assigned pin numbers on the connector. These assignments are given in the following table:

pin	name
1	Protective ground
2	Transmitted Data
3	Received Data
4	Request to Send
5	Clear to Send
6	Data Set Ready
7	Signal Ground (Common)
8	Received Line Signal Detect (Data Carrier Detect)
9	Reserved for testing
10	Reserved for testing
11	Unassigned
12	Secondary Received Line Signal Detect
13	Secondary Clear to Send
14	Secondary Transmitted Data
15	Transmission Signal Element Timing
16	Secondary Received Data
17	Receiver Signal Element Timing
18	Unassigned
19	Secondary Request to Send
20	Data Terminal Ready
21	Signal Quality Detector
22	Ring Indicator
23	Data Signal Rate Detector
24	Transmit Signal Element Timing
25	Unassigned

A number of these circuits are unassigned, concerned with synchronous transmission, or have only a secondary control function and are rarely used. The following table gives the nine remaining RS232 circuits relevant to asynchronous communication on microcomputers.

pin	abrv.	name	direction	function
1	---	protective ground	-----	safety ground
2	TD	transmitted data	to DCE	outbound DTE data
3	RD	received data	to DTE	inbound DTE data
4	RTS	request to send	to DCE	DTE wants to transmit data
5	CTS	clear to send	to DTE	DCE is ready to receive data
6	DSR	data set ready	to DTE	DCE is ready to communicate with DTE
7	---	signal common	-----	common line for circuits
8	DCD	data carrier detect	to DTE	data link in process
20	DTR	data terminal ready	to DCE	master modem enable

The data being transferred is carried on only two circuits, pins 2 and 3, transmitted data and received data. The absolute minimum RS232 system would only require these two connections and a common ground (and some would argue that life would be simpler if the standard was maintained at that level). All the other lines are control circuits intended to control the flow of data. The control circuits should be interpreted by the devices at each end as signals to start or stop the exchange of data.

15.1.2 Functional Descriptions of Interchange Circuits

The following paragraphs briefly describe the formal EIA definitions of each circuit together with some additional comments on the use of these circuits.

Protective Ground (pin 1)
This pin should be connected to ground (earthed preferably). The DIN plug metal shield is a good ground connection at the BBC end and when using shielded cable the woven wire screen should be used to connect the two protective grounds.

Signal Common (pin 7)
This is another zero volt connection. This circuit provides the common return for all circuits and thus completes the circuit and allows the current to flow. This circuit should not normally be connected to the protective ground pin 1.

Request to Send (RTS, pin 4)
Clear to Send (CTS, pin 5)
RTS officially 'conditions' the modem for transmission. This signal is an output from the computer (DTE) to the modem (DCE). RTS/CTS handshaking was intended for use in a half-duplex situation (i.e. when data transmission may only occur in one direction at any one time). The computer keeps RTS inhibited while data is being received from the modem. When the computer wishes to transmit data to the modem it signals the intention using the RTS line. The modem will not respond instantly but will signal its readiness to receive via the CTS input of the computer. The computer should not begin transmission to the modem until CTS has been asserted by the modem.

In the real world most communication is full-duplex (i.e. transmission in both directions can occur simultaneously). Most modems permanently assert CTS or tie it to Data Carrier Detect (pin 8)

The two handshaking lines on the BBC micro are called RTS and CTS and while these names are perfectly adequate in a purely semantic sense

they are misnamed in terms of the EIA standard. A description of how the BBC micro uses these signals is given in section 15.2.

Data Set Ready (DSR, pin 6)
DSR is asserted by modems when a communication channel is open. This normally means that entry has been made into the telephone system. The precise meaning depends on whether the modem is in originate or answer mode.

Usually this signal is permanently asserted by microcomputer modems and is used as a signal to indicate that the modem is connected and powered up.

Data Carrier Detect (DCD, pin 8)
Otherwise known as Received Line Signal Detect, this signal is asserted when the modem has detected a remote carrier. This normally means that contact has been made with another modem. The signal remains asserted for the duration of the link.

The loss of this signal in the middle of a session indicates that the modem is still connected and functioning, but that contact with the remote modem has been lost.

Data Terminal Ready (DTR, pin 20)
This signal informs the modem that the computer is ready for communication. In the words of the standard it 'prepares the DCE to be connected and maintains the connection established by external means'. Most modems will not receive data unless this signal is asserted. The signal is, in effect, a master control.

This signal generally indicates that the computer is connected and powered up.

Transmitted Data (TD, pin 2)
This circuit carries the data transmitted from the computer to the modem. Data is not transmitted unless the following circuits are asserted:

1. RTS
2. CTS
3. DSR
4. DTR

The CTS and RTS requirements make no sense in full-duplex modems where these signals have no meaning.

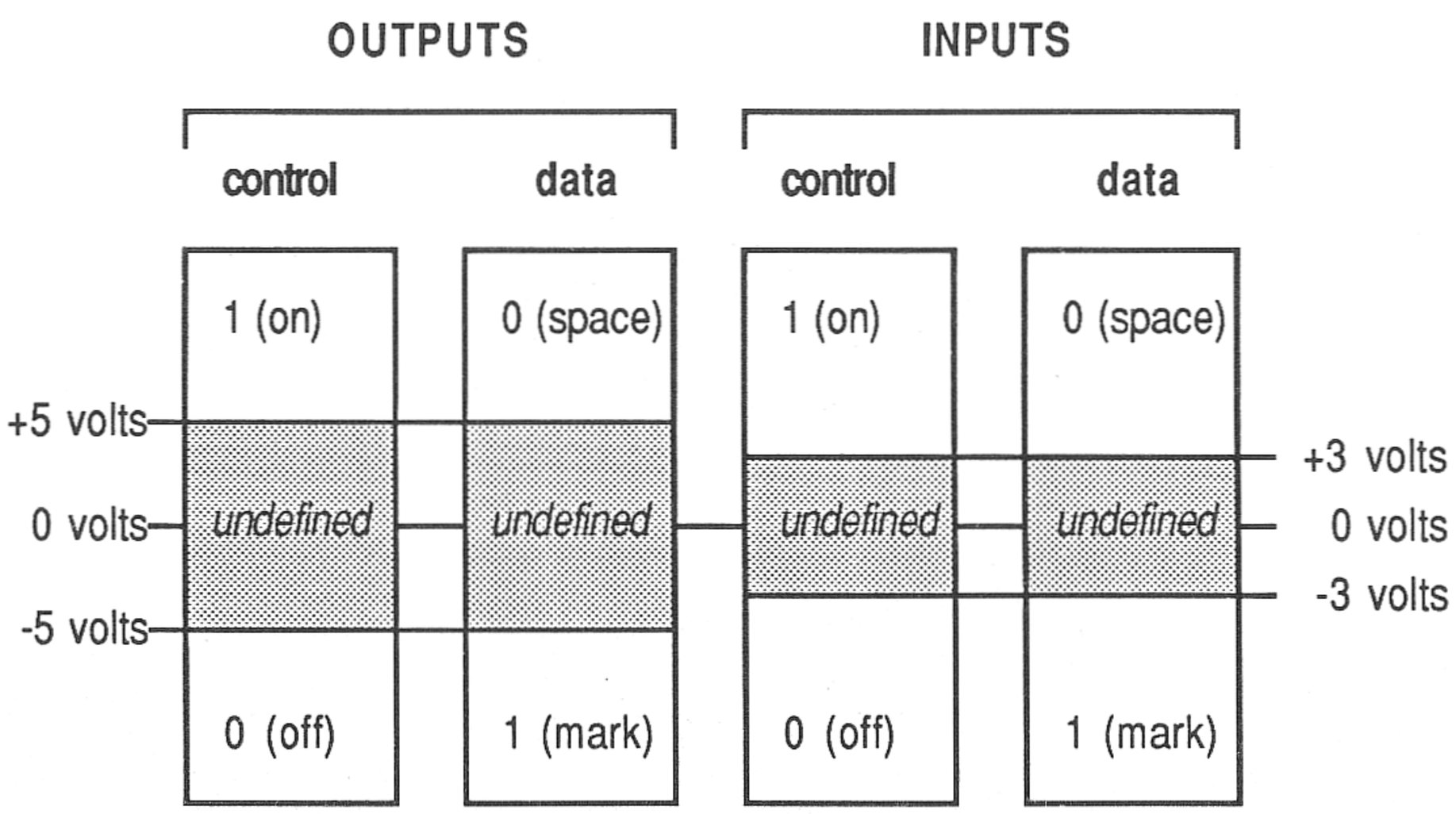
Received Data (RD, pin 3)
The activity of this line is not dependent directly on any other signal.

15.1.3 Electrical Signal Characteristics

Although the following points will only be of peripheral interest to most programmers, they are included here to complete this outline of the RS232 standard.

The speed of transmission is defined within the standard from zero to a nominal upper limit of 20,000 bits per second. For most devices this limit tends to be 19,200 baud. The standard also cautions against cable lengths in excess of 50 feet unless the total capacitance is less than 2,500 picofarads.

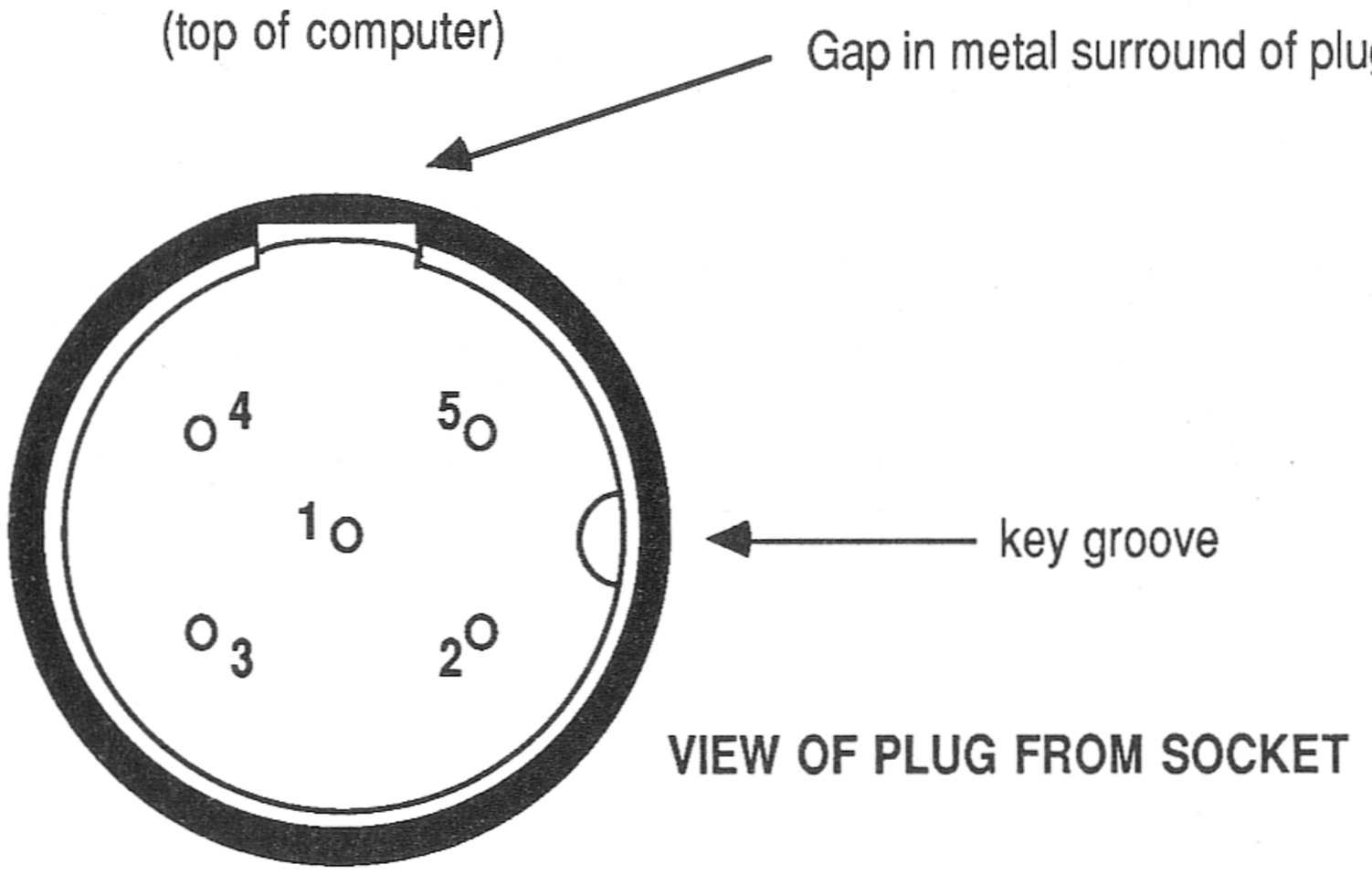
The standard also states that the interface must be able to sustain a short circuit of infinite duration between any two of its pins and in such a case the current must not exceed 500 mA.



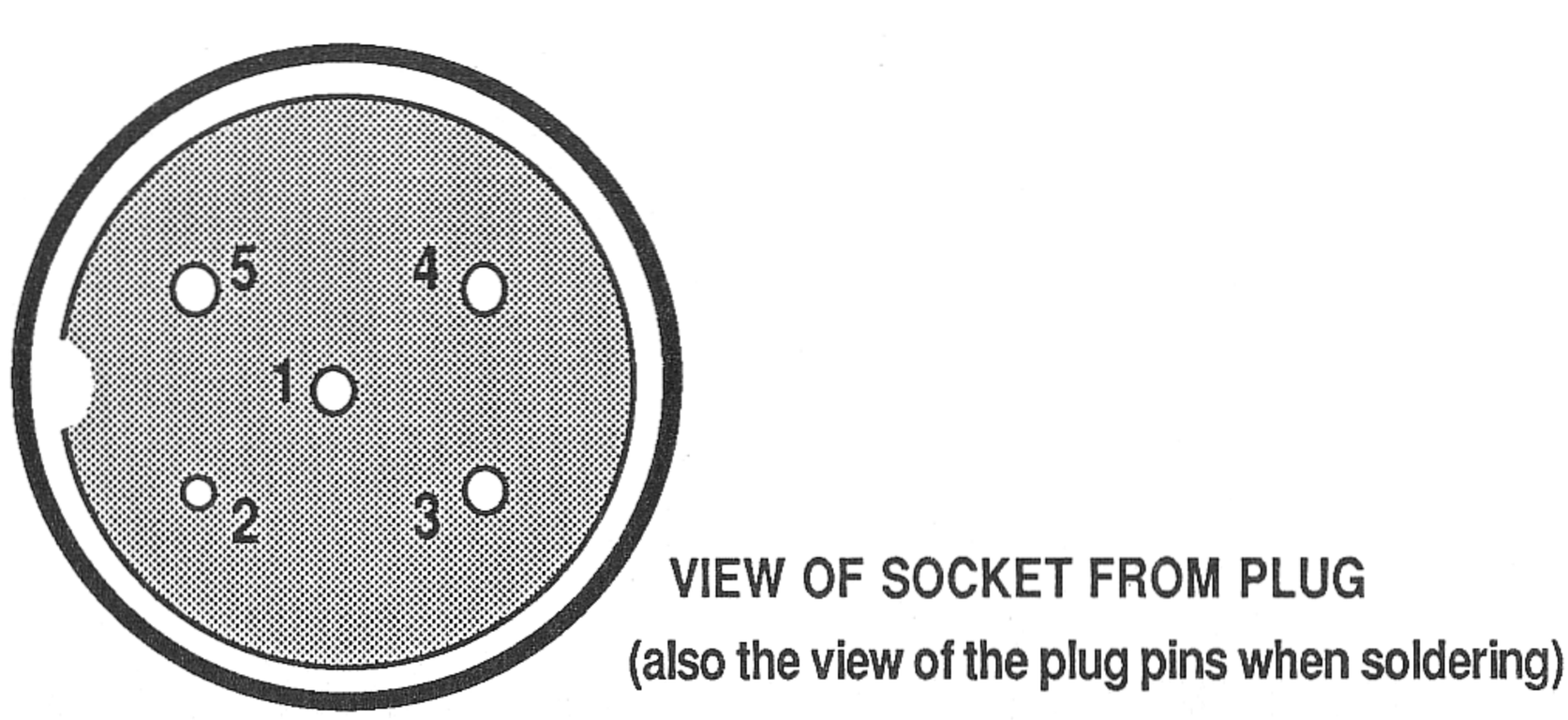
The RS232 standard specifies the voltages in terms of their magnitude and polarity. No voltage should exceed ±15 volts. The diagram above indicates the logic levels for the control and data signals.

15.2 The Acorn RS423/RS232 implementation

Acorn use a five connection serial link. A table of the signals used and a diagram showing their connections in the RS423/232 socket are illustrated below.



1	signal ground
2	clear to send
3	transmited data
4	received data
5	request to send



One major complaint against the infamous Acorn RS423/RS232 connector is the fact that it is possible to insert it in two different orientations. The plug may be inserted with the notch in the metal shield facing the top of the computer case (usually accepted as the correct position) or 180º rotated with the notch facing the bottom of the case. Although no harm is normally done by inserting the plug the wrong way round, there is an inadequacy in the design. One of the first things to test in the event of a failure in the serial port is that the plug has been inserted the right way round and that the cable has been wired up correctly.

In comparison with many other manufacturers (most notably IBM and compatibles) Acorn have been exceedingly generous by including an operating system which provides interrupt driven buffered serial I/O with hardware handshaking.

The use of the two data signals is straight-forward, but in comparison with other serial interfaces there is a dearth of control signals. In essence the Acorn design is a model of simplicity. The control signals consist of a Request to Send output signal and a Clear to Send input signal. The RTS line is asserted until the serial input buffer becomes full and is re-asserted when the buffer empties again. The transmission of characters is dependent on the presence of an asserted CTS line. If this signal is lost then transmission is halted and only recommences when the CTS signal is re-asserted. Clearly the RTS and CTS lines are complementary and connecting two BBC micros together via the serial port can be carried out by connecting the RTS and TD of each micro to the CTS and RD of the other.

This use of RTS and CTS corresponds more closely with the Data Terminal Ready and Data Set Ready signals of the EIA standard than with the true RTS and CTS definitions, although it may be argued otherwise.

Problems often arise when attempting to connect a BBC micro to a peripheral device or another computer possessing a 25 pin connector. Although Acorn are not the only manufacturers to use RTS and CTS as the handshaking lines, it is rarely the case that just connecting these two signals will be sufficient. Because of the wide variation in various RS232 implementations there are no golden rules for connecting different devices together so that they always work. Several hardware devices are available which act as protocol converters (like the Adder range of *Peripheral Managers*!) which can sometimes overcome the direct interfacing problems.It is hoped that the information given above will help anyone attempting to interface their BBC microcomputer with another computer or peripheral via the serial port.

15.3 OS calls for using the serial port

All the following calls relate directly to the RS423/RS232/Serial system. The majority of these calls will not be implemented on the unexpanded Electron and will be passed on to the paged ROMs as unknown OSBYTE calls.The Master Compact requires the optional interface chips to be inserted before its RS232 port can be used.

15.3.1 Select input stream OSBYTE call

Call address &FFF4
Indirected through &20A
A=&02 (*FX2)

Entry parameters:

X determines input device(s)

X=0	keyboard selected, RS232 disabled (default state)
X=1	RS232 selected and enabled, keyboard disabled
X=2	Keyboard selected, RS232 enabled

On exit,

X=0, if the keyboard was the previous input source
X=1, if the serial port was the previous input source

A is preserved, other registers undefined.

15.3.2 Select output stream OSBYTE call

Call address &FFF4
Indirected through &20A
A=&03 (*FX3)

Entry parameters:

X determines output device(s),Y=0

bit	action if set
0	Enables RS232 output
1	Disables VDU driver
2	Disables printer output
3	Enables printer, independent of Ctrl-B or Ctrl-C
4	Disables spooled output
5	not used
6	Disables printer output unless character is preceded by a VDU 1 (or equivalent)
7	not used

*FX3,0 selects the default output options which are:
RS423 disabled
VDU enabled
Printer enabled (output controlled by VDU 2/3 or Ctrl-B/C)
Spooled output enabled (requires *SPOOL command)

The OSBYTE call with A=&EC (236) can also be used to read and write the output stream selection byte. This OSBYTE call has the advantage that the Y register can be used as a bit-mask to avoid writing those bits which are not to be changed.

On exit,

X=old output stream status byte

A is preserved, other registers undefined.

15.3.3 Set serial receive rate OSBYTE call

Call address &FFF4
Indirected through &20A
A=&07 (*FX7)

Entry parameters:

X determines baud rate

X=1	75	baud receive
X=2	150	baud receive
X=3	300	baud receive
X=4	1200	baud receive
X=5	2400	baud receive
X=6	4800	baud receive
X=7	9600	baud receive
X=8	19200	baud receive

On exit:

A is preserved
X & Y contain old serial ULA register contents (not Electron)
C is undefined

15.3.4 Set serial transmission rate OSBYTE call

Call address &FFF4
Indirected through &20A
A=&08 (*FX8)

Entry parameters:

X determines transmission rate

X=1	75	baud transmit
X=2	150	baud transmit
X=3	300	baud transmit
X=4	1200	baud transmit
X=5	2400	baud transmit
X=6	4800	baud transmit
X=7	9600	baud transmit
X=8	19200	baud transmit

On exit:

A is preserved
X & Y contain old serial ULA register contents (not Electron)
C is undefined

15.3.5 Read/write RS423 mode OSBYTE call

Call address &FFF4
Indirected through &20A
A=&B5 (181)

This call is implemented on all Acorn-BBC computers. On the unexpanded Electron this call will have no effect unless a suitable hardware and software expansion has been added to implement RS423.

There are two RS423 modes available. The default mode treats RS423 input differently from keyboard input in that if an ESCAPE character (&1B normally) is received, an ESCAPE is not generated and instead the ASCII character is placed in the input buffer. This has been provided to simplify the implementation of such things as terminal emulators. The second mode treats RS423 input just as if it had come from the keyboard.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag value is returned in X.

Flag=0	ESCAPEs are recognised soft keys are expanded character entering input buffer event generated cursor editing performed

Flag=1 All characters enter input buffer (default)
character entering buffer event not generated

15.3.6 Read/write RS423 use flag OSBYTE call

Call address &FFF4
Indirected through &20A
A=&BF (191)

This OSBYTE is used on all computers, but is reserved for expansion software on the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag value is returned in X.

bit 7 set - RS423 free.
bit 7 clear - RS423 busy
bits 0 to 6 - undefined.

15.3.7 Read/write 6850 control register OSBYTE call

Call address &FFF4
Indirected through &20A
A=&9C (156)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old register value is returned in X.

This call may be used to change or read the 6850 ACIA control register. This call updates the operating system's RAM copy of this register at the same time.

The control register bits have the following meanings:

Bit 0	Bit 1	Effect
0	0	divide counter by 1
1	0	divide counter by 16
0	1	divide counter by 64 (default for RS423)
1	1	master reset

This counter is used to set the cassette system baud rates.

Bit 2	Bit 3	Bit 4	Effect
0	0	0	7 bit word, even parity, 2 stop bits
1	0	0	7 bit word, odd parity, 2 stop bits
0	1	0	7 bit word, even parity, 1 stop bit
1	1	0	7 bit word, odd parity, 1 stop bit
0	0	1	8 bit word, 2 stop bits
1	0	1	8 bit word, 1 stop bit
0	1	1	8 bit word, even parity, 1 stop bit
1	1	1	8 bit word, odd parity, 1 stop bit

These bits control the RS423 data format.

Bit 5	Bit 6	Effect
0	0	RTS low, transmit interrupt disabled
1	0	RTS low, transmit interrupt enabled
0	1	RTS high, transmit interrupt disabled
1	1	RTS low, break level on data output, transmit interrupt disabled

These bits control the level of the RTS line, the interrupt generated by the transmit data register empty state, and the break level on data output.

Bit 7, when set, enables the receive data register full, over-run, or DCD transition interrupts.

For a more detailed description of this register refer to the manufacturer's data sheet for the 6850 ACIA.

Read OS copy of 6850 control register OSBYTE call

Call address &FFF4
Indirected through &20A
A=&C0 (192)

This OSBYTE is used on all computers, but is reserved for expansion software on the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

This call is equivalent to OSBYTE &9C except that it does not update the register in the 6850. This call should not be used to write the control flag as it would cause the operating system RAM copy to become inconsistent with the 6850 register contents.

15.3.8 Read/write RS423 handshake level OSBYTE call

Call address &FFF4
Indirected through &20A
A=&CB (203)

This OSBYTE is used to store the interrupt mask for the ULA on the Electron (see interrupts chapter 8).

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old handshake level is returned in X.

This OSBYTE controls the space remaining in the RS423 input buffer when RS423 input is halted by the operating system entering a buffer full state (which sets the RTS line high). The default value is 9. The free space remaining in the buffer allows extra characters to be received before buffer overflow occurs and data is lost. The value selected should reflect the response time at the transmission end and the time taken for the operating system to act upon the buffer full situation.

15.3.9 Read/write RS423 input suppression flag OSBYTE call

Call address &FFF4
Indirected through &20A
A=&CC (204)

On the Electron this call is used to access the firm key pointer (see keyboard calls, chapter 14).

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag value is returned in X.

If this location contains 0 then RS423 input is accepted otherwise RS423 input is ignored (RS423 receive errors will still cause an event).

15.3.10 Read/write RS423/cassette flag OSBYTE call

Call address &FFF4
Indirected through &20A
A=&CD (205)

On the Electron this call is used to access the current firm key string length.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag value is returned in X.

If flag=0 RS423 data passed to serial port.
If flag=&40 RS423 data passed to cassette port.

Any change is only effected following a baud rate selection using OSBYTE calls &07 or &08.

15.3.11 Read/write IRQ mask for 6850 OSBYTE call

Call address &FFF4
Indirected through &20A
A=&E8 (232)

This location is reserved for expansion on the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old mask value is returned in X.

This location contains a software copy of the interrupt bit mask for the serial communications chip (6850).

Default value &FF.

Refer to the interrupts (chapter 8) for further information about 6850 interrupts.

15.3.12 Read OS copy of serial ULA register OSBYTE call

Call address &FFF4
Indirected through &20A
A=&F2 (242)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

This location contains a software copy of the value currently written to the serial processor ULA register. This call should not be used for writing as it would make the copy of the register contents different from the actual register contents.

All the serial ULA functions can be controlled with :-

OSBYTE with A=&89/*FX 137	motor control
OSBYTE with A=&CD/*FX 205	cassette/RS423 select
OSBYTE with A=&7,&8/*FX 7,8	RS423 baud rate control

This location is reserved for future Acorn expansion on the Electron.

The serial ULA consists of a single register at &FE10. This register should not be programmed directly. All the facilities offered by this chip are available from operating system calls. This brief description is included in the absence of a conventional manufacturers data sheet. The serial ULA control register is 8 bits wide and is write only.

Bits 0 to 2 - define the transmit baud rate. Bits 3 to 5 - define the receive baud rate

bit 5	bit 4	bit 3		bit 2	bit 1	bit 0	reg. bits
Transmit				Recieve			Baud rate
0	0	0		0	0	0	19200
1	0	0		1	0	0	9600
0	1	0		0	1	0	4800
1	1	0		1	1	0	1200
0	0	1		0	0	1	300
0	1	1		0	1	1	150
1	1	1		1	1	1	75

These baud rates rely on the 6850 control register being set to divide the clock rate by 64.

This is the only way that the currently set baud rates may be read. The relationship between the 4 bit register values and the baud rate numbers used by OSBYTEs &07 and &08 is that the ULA value inverted and plus 1 gives the OSBYTE baud rate value.

Bit 6 - if set the RS423 system has control of the serial system otherwise the cassette system is in use.
Bit 7 - if set the cassette motor and relay are switched on.

15.3.13 Read/write CFS timeout counter OSBYTE call

Call address &FFF4
Indirected through &20A
A=&B0 (176)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old counter value is returned in X.

This counter is decremented once every vertical sync. pulse (50 times per second). The timeout counter is used to time inter-block gaps and leader tones.

16 Filing System Implementations

This chapter consists of brief descriptions of a number of different filing systems which have been implemented on the Acorn BBC series microcomputers. The Tape and *ROM filing systems are incorporated into the operating system software itself. Other filing systems reside in their own paged ROMs (on Master series computers ADFS and DFS are already installed within the 1 megabit ROM). Additional hardware is also required for the use of most filing systems.

16.1 Filing system calls

In their design of the operating system software for the BBC microcomputers, Acorn have defined a number of standard functions which have to be supported by filing system software. Two filing systems are incorporated into the operating system itself. These are the cassettte filing system and the ROM filing system. Other filing systems software is implemented in paged ROMs to support expansion hardware such as floppy disc drives and teletext adapters. Each new filing system adheres to the filing system rules so that all other software using the filing system is unaffected by any change in filing systems which may occur.

The complete filing system specification is designed cope with all types of storage media from cassette tapes to winchester hard discs. Some filing systems do not implement the full filing system specification due to the physical limitations of the media they use. For example write operations are not supported by the ROM filing system and the cassette filing system does not allow the use of sub-directories.

The following sections describe the standard filing system calls with notes on how the different filing systems implement them.

16.1.1 OSFILE

Read or write a whole file or its attributes

Call address &FFDD
Indirected through &212 (FILEV)

This routine is used to load or save whole files.

On entry:

A specifies action to be performed
X and Y contain the address of a parameter block

XY +	0 1	address pointing to filename + CR (LSB first)
	2 5	file load address
	6 9	file execution address
	10 13	start address or length
	14 17	end address or file attributes

The file attributes consist of a file attribute byte (shown in the table below) which is stored in XY+14, and for the NFS a two byte date attribute. The date attribute consists of the day of the month in the first byte, the month in the least significant 4 bits of the next byte, and the year - 81 stored in the remaining 4 bits.

bit	attribute	meaning when set	meaning when 0
0	R attribute	file cannot be read	file can be read
1	W attribute	file cannot be written	file can be written
2	E attribute	file is 'execute only'	attribute not set
3	L attribute	file can't be deleted	file not locked
4	NFS use	private use only	readable by others
5	NFS use	private use only	writable by others
6	not used		
7	not used		

Action of call with various values in A:

0	save block of memory returning file length and attributes
1	write catalogue information for named file
2	write load address for named file
3	write execution address for named file
4	write attributes for named file
5	read catalogue information
6	delete named file, returning catalogue information
7	create empty file of defined size
255	load named file, if XY+6 contains 0, use specified address

The call with A=7 is new to the Master series computers.

On exit:

The parameter block is updated by some calls.
A is undefined except after the call with A=5
A=0 if file not found
A=1 if file found
A=2 if a directory was found
A=&FF if E attribute set (ADFS only)
X and Y are preserved
Other registers are undefined

Interrupts may be enabled by this call.

16.1.2 OSARGS

Read or write a file's attributes

Call address &FFDA
Indirected through &214 (ARGSV)

On entry :

A specifies action to be taken
X points to a 4 byte area in zero page (always i/o processor)
Y contains file handle or 0

Entry parameters			Action
A	Y reg.	X reg.	
0	0		return filing system number in A
	file handle	ptr to ZP	return sequential file pointer in zero page
1	0	ptr to ZP	return address of remaining command line in zero page
	file handle	ptr to ZP	write sequential file pointer from zero page
2	file handle	ptr to ZP	return length of file in zero page
255	0		write any buffered data to all pending files
	file handle		write any buffered data to specific file

On exit:

X and Y are preserved

all other registers undefined (except when A=0, Y=0)

Filing System Numbers	
0	no filing system selected
1	1200 baud CFS
2	300 baud CFS
3	ROM filing system
4	Disc filing system
5	Network filing system
6	Teletext filing system
7	IEEE filing system
8	ADFS
9	Host filing system
10	Videodisc filing system

Interrupts may be enabled by this call

16.1.3 OSBGET

Read a single byte from an open file

Call address: &FFD7
Indirected through &216 (BGETV)

On entry:
Y contains file handle, as provided by OSFIND

On exit:
A contains the byte read from the file
(the sequential file pointer is incremented by one)
C is set if the end of the file has been reached
X and Y are preserved
Other register values are undefined
Interrupts may be enabled

16.1.4 OSBPUT

Write a single byte to an open file

Call address: &FFD4
Indirected through &218 (BPUTV)

On entry:
A contains byte to be written
Y contains file handle, as provided by OSFIND

On exit:

A, X and Y are preserved
Other register values are undefined
Interrupts may be enabled

Fast Tube BPUT OSBYTE call

Call address &FFF4
Indirected through &20A
A=&9D (157)

This call is a faster alternative to OSBPUT when used from a second processor.

On entry:

X contains byte to be written
Y contains file handle

On exit:

X and Y are undefined.

16.1.5 OSGBPB

Read or write multiple bytes to an open file

Call address: &FFD1
Indirected through &21A (GBPBV)

On entry:

A specifies action to be performed
X and Y contain the address of a parameter block

XY +	0	file handle (or disc cycle number, for A=8)
	1 4	start address of data (LSB first) (or address for returned data)
	5 8	number of bytes to transfer (LSB first) (or no. of filenames for call with A=8)
	9 12	sequential pointer value to be used (LSB first)

The various actions of the OSGBPB call are summarised in the following table.

A	OSGBPB call Action
1	write bytes to file at sequential file pointer specified
2	append bytes to file at current file pointer
3	read bytes from specified position in file
4	read bytes from current position in file
5	read title, option and drive to memory
6	read current directory and drive names
7	read current library and drive names
8	read file names from the current directory

parameter block	
0	length of title - n
1	title string or directory
n+1	start option
n+2	drive number

0	drive string length - x
1	drive string
x+1	directory name len. - y
x+2	directory string

0	length of 1st file name - x
1	1st filename
x+1	length of 2nd file name - y
x+2	2nd filename
y+x+3	length of next file name
	repeated as requested

results stored in memory specified in parameter block

On exit:

A, X and Y are preserved
If the carry flag is clear in the event of a successful transfer
C=1 if the transfer could not be completed.

In the event of a transfer not being completed the parameter block contains the following information:

(a) The number of bytes or names not transferred in the *number of bytes to transfer* field.
(b) The *address* field contains the next location of memory due for transfer.
(c) The *sequential pointer* field contains the sequential file pointer value indicating the next byte in the file due for transfer.

16.1.6 OSFIND

Open or close a file for random or sequential access

Call address &FFCE
Indirected through &21C (FINDV)

On entry:

A specifies action to be taken
X and Y contain relevant parameters (see table below)

Entry parameters		Action
A	X and Y registers.	
0	Y=0	all files to be closed
	Y=file handle	specific file to be closed
64	XY=addr. of filename	file to be opened for input, file handle returned in A
128	XY=addr. of filename	file to be opened for output, file handle returned in A
192	XY=addr. of filename	file to be opened for random access, handle ret. in A

On exit:

A is preserved following a call with A=0
A returns the file handle following successful file opening
If a file could not be opened then A is returned containing 0
X and Y are preserved
other registers are undefined

When opening a file for output a file will be created if one does not exist (default length 16k - DFS, 64k - ADFS). If a file already exists the sequential file pointer will be set to the start of the file.

Note that if the file could not be opened because the filename was syntactically incorrect, or involves a non-existent directory, a BRK may be executed with an error message 'Not found error'.

16.1.7 FSCV

The filing system control vector (&21E on model Bs, or in Hazel on Master series computers)

This vector is used by the operating system to invoke a number of miscellaneous filing system functions.

A	action	X & Y entry parameters	exit parameters
0	*OPT command	*OPT parameters	
1	EOF being checked	X=file handle	X=&FF if EOF,X=0 otherwise
2	*/ command	XY point to command line	
3	unrecognised * command	XY point to command line	
4	*RUN command	XY point to command line	
5	*CAT command	XY point to command line	
6	New filing system start		
7	return file handle range		X=minimum, Y=maximum
8	OS has received a *command		
9	*EX command	XY point to command line	
10	*INFO command	XY point to command line	
11	*RUN command for library	XY point to command line	

16.1.8 General filing system OSBYTE calls

The following OSBYTE calls have general filing system effects.

Read machine high order address OSBYTE call

Call address &FFF4
Indirected through &20A
A=&82 (130)

This OSBYTE call provides the 16 bit high order address for filing system addresses which require 32 bits (see section 18.1).

No entry parameters

On exit:
X and Y contain the high order address (LSB, MSB).

Close any *SPOOL/*EXEC files OSBYTE call

Call address &FFF4
Indirected through &20A
A=&77 (119)

This call causes the closure of any open *SPOOL or *EXEC files. A paged ROM service call with A=&10 is also generated (see section 17.5.1)

On exit:
A is preserved
All other registers are undefined.

Read or write *EXEC file handle OSBYTE call

Call address &FFF4
Indirected through &20A
A=&C6 (198)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

The location used by this OSBYTE contains the file handle used for the *EXEC file.

Read or write *SPOOL file handle OSBYTE call

Call address &FFF4
Indirected through &20A
A=&C7 (199)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

The location used by this OSBYTE contains the file handle used for the *SPOOL file.

Select filing system options (*OPT) OSBYTE call

Call address &FFF4
Indirected through &20A
A=&8B (139)

Entry parameters:
X contains the option number (1st *OPT parameter)
Y contains the option value (2nd *OPT parameter)

On exit:
A is preserved
All other registers are undefined.

Check for EOF OSBYTE call

Call address &FFF4
Indirected through &20A
A=&7F (127)

On entry:
X contains the file handle of the file to be checked.

On exit:

X<>0 if the end of the file has been reached.
X=0 if the end of the file has not been reached.
A and Y are preserved
All other registers are undefined.

16.2 Master Series Filing Systems

Auxilary filing systems

The Master series computers have a number of additional filing system facilites controlled by a filing system handler.

The filing system handler is a part of the operating system on these machines. It intercepts all calls relevant to filing systems and then feeds these calls to the appropriate filing system. This approach enables more than one filing system to be easily accessible at the same time.

The default filing system is determined by either the power-up configuration or by the use of a command such as *DISC or *TAPE. A secondary filing system may be specified which is used only when disc based commands are not found on the default filing system. This filing system is called the library filing system (it is intended that it should be used to house a library of regularly used programs). A third type of filing system called a temporary filing system is also available. Temporary filing systems are invoked by specifying the target filing system as a prefix for the filing system command. The prefixes used consist of the filing system name (DFS, ADFS, NET, TAPE etc) with a hyphen before and after it.

for example

```
*LOAD -DISC-prog
```

specifies that the file, prog, should be loaded from the disc filing system.

The filing system handler intercepts filing system calls that are made to the relevent addresses in page &FF. This means that directly using the contents of the filing system call vectors will bypass this mechanism. The filing system handler code places an address to one of its own routines into the filing system control vector and maintains its own copy of the active filing system control register for the current filing system.

The additional workspace required for the filing system handler routines is provided by the additional sideways memory in page &DF (Hazel). The use of this memory is illustrated in the following diagram.

Page &DF (Hazel) Allocations

&DF00	current filing system number
&DF01	active filing system number
&DF02	library filing system number
&DF03	current filing system ROM number
&DF04 &DF05	command line pointer, transient commands
&DF06 &DFC1	upto 17 filing system information blocks (11 bytes each) terminated by a zero byte
&DFC2	filing system flags
&DFC3 &DFC4	BCD line number for BUILD/APPEND/LIST
&DFC5	last char printed by BUILD/APPEND/LIST
&DFC6	temporary filing system flag
&DFC7 &DFD3	OSGBPB for destination of *MOVE
&DFD4	source handle for *MOVE
&DFD5	destination handle for *MOVE
&DFD6	MSB of *MOVE buffer address
&DFD7	*MOVE buffer length in pages
&DFD8 &DFD9	destination name pointer for *MOVE
&DFDA &DFDB	copy of FSCV for active filing system
&DFDC	copy of ACCCON during *MOVE
&DFDD	ACCCON changed flag (0 if not changed)
&DFDE &DFFF	unused

0 7	filing system name
8	minimum file handle
9	maximum file handle
A	filing system number

bit 7	use ASCII only when printing or APPEND/BUILD flag
bit 6	no line numbers when printing

Make temporary filing system permanent OSBYTE call

Call address &FFF4
Indirected through &20A
A=&6D (109)

This call, when issued while a temporary filing system is active, makes the temporary filing system become permanent.

16.3 The main filing systems

A number of different filing systems have been produced by Acorn or are supported by them. The cassette and ROM filing systems are available on all the Acorn BBC series machines in their unexpanded state, whereas the IEEE filing system and the Teletext filing system (for example) are only available with specific expansion hardware.

Filing System	no.	BREAK	temp fs name	*command	worksp. p	worksp. a
no filing system selected	0					
1200 baud CFS	1	SPACE	-TAPE- or -CFS-	*TAPE	0	0
300 baud CFS	2		-TAPE- or -CFS-	*TAPE 3	0	0
ROM filing system	3	SHIFT-SPACE	-ROM-	*ROM	0	0
Disc filing system	4	D	-DISC-	*DISC or *DISK	9	2
Network filing system	5	N		*NET	2	2
Teletext filing system	6	T		*TELESOFT	23	
IEEE filing system	7			*IEEE	2	1
ADFS	8	A or F	-ADFS-	*ADFS or *FADFS	6	1
Host filing system	9					
Videodisc filing system	10	Q or L		*VFS or *LVFS		

The workspace column indicates the number of 256 byte pages which are claimed by the filing systems as private or absolute workspace (see section 17.5.1).

16.3.1 The Cassette Filing System

Magnetic tape recorders are a relatively inflexible device for the storage of data. Reflecting the limitations of the storage medium, the cassette filing system can only implement a sub-set of the filing system calls. Variations from the standard filing system specification are :

OSFILE - only load and save operations are supported
OSARGS - only filing system identification supported
OSGBPB - not implemented except on the Master where A=2 and A=4 are implemented.
OSFIND - only input and output supported (up to 2 files)
OSFSC - calls 0 to 6 are implemented, call 3 returns a Bad command error.
File handles given are always 1 for input and 2 for output.

The following OSBYTEs are implemented by the cassette filing system:

Switch cassette motor relay OSBYTE call

Call address &FFF4
Indirected through &20A
A=&89 (137)

This OSBYTE call is the equivalent of the *MOTOR command.

On entry:

X=0 relay off
X=1 relay on

On exit:

X and Y are undefined.

Select CFS OSBYTE call

Call address &FFF4
Indirected through &20A
A=&8C (140)

This OSBYTE call is the equivalent of the *TAPE command. Following this call the cassette filing system is selected; the baud rate is specified by the X value.

On entry:

X=0	default baud rate (1200)
X=3	300 baud
X=12	1200 baud

On exit:

X and Y are preserved.

Read or write CFS/RFS switch OSBYTE call

Call address &FFF4
Indirected through &20A
A=&B7 (183)

This OSBYTE call is used to read or write the flag used to determine selection of either the cassette filing system or the ROM filing system. This switch is required because the two filing systems use some common code.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old status value is returned in X.

This location contains 0 for *TAPE and 2 for *ROM, all other values are meaningless.

The cassette tape format

offset	*description*	*length*
	1st BLOCK HEADER	
0	&2A, a synchronisation byte	1
1	file name (1-10 chars.)	n
1+n	&00, a file name terminator	1
2+n	load address (lo byte first)	4
6+n	execution address (lo byte first)	4
10+n	block number (lo byte first)	2
12+n	block length (m, in bytes)	2
14+n	block flag	1
15+n	4 bytes - unused	4
19+n	header CRC (1 to 18+n inclusive)	2
	BLOCK DATA	
21+n	data (0 - 256 bytes)	m
21+n+m	data block CRC	2

bit	meaning
0	*RUN only
1	not used
2	not used
3	not used
4	not used
5	not used
6	no data
7	last block

Cyclic Redundancy Check (CRC) values are a way of generating a unique fingerprint for a block of data. If the data becomes corrupted then the CRC value stored with the data will not match the CRC value calculated from the data recovered from the storage media. A routine for calculating CRCs is given in section 17.5.7 as part of the ROM filing system example.

16.3.2 The *ROM filing system

An account of this filing system is given in section 17.5. Variations from the standard filing system are:

OSFILE - only load operations are supported
OSARGS - only filing system identification supported
OSBPUT - not implemented
OSGBPB - not implemented except on the Master where A=4 is implemented.
OSFIND - only input is supported (only 1 file)
OSFSC - calls 0 to 6 are implemented, call 3 returns a Bad command error.
File handle given is always 3 for input only.

Select RFS OSBYTE call

Call address &FFF4
Indirected through &20A
A=&8D (141)

This OSBYTE call is the equivalent of the *ROM command.

No entry parameters.

On exit:

X and Y are undefined.

The *ROM data format is given in section 17.5.6

16.3.3 Acorn DFS

The disc filing system differs from the standard filing system in the following ways.

The file attributes accessed using the OSFILE call are represented by a single bit. This bit is called the lock flag. A file is locked if either the 'you cannot write' or the 'you cannot delete' bits are set when the file attributes are written using the OSFILE call.

The OSFILE call never returns values of 0 or 2; instead a 'File not found' error is returned.

Directory names consist of a single character and file names are up to 7 characters long.

The media title is the disc title, and is up to 12 characters in length.

OSWORD calls with A=&7D, &7E and &7F are also implemented by the DFS.

The DFS catalogue format

The catalogue information on a disc is contained in the first two sectors of each disc surface. The catalogue contains information which identifies the disc and information about each file on the disc (up to a total of 31 files).

The disc information consists of:

A file name
The file's directory
File locking information
A load address
An execution address
The file's length (in bytes)
The file's sector address on disc

The actual layout of the catalogue information is designed to make optimum use of the space and to enable the DFS to access files as rapidly as possible. The DFS uses 10 bit addresses which do not fit neatly into ordered bytes. This leads to the most significant 2 bits of these addresses being stored remotely from the least significant two bytes.

The format of the catalogue sectors is :

Sector 0

offset	description
&00 &07	First 8 bytes of the 12 byte disc title
&08 &0E	1st filename
&0F	Directory of 1st file + file access flag (bit 7)
&10 &1E	2nd filename
&1F	Directory of 2nd file + file access flag (bit 7)
&F8 &FE	31st filename
&FF	Directory of 31st file + file access flag (bit 7)

Sector 1

&00 &03	Last 4 bytes of 12 byte disc title
&04	Cycle no. (BCD count of catalogue writes)
&05	Number of catalogue entries*8
&06	boot option + sector information
&07	Total number of sectors on disc (LSB)
&08 &09	1st file's load address (least sig. 16 bits)
&0A &0B	1st file's execution address (least sig. 16 bits)
&0C &0D	1st file's length in bytes (least sig. 16 bits)
&0E	Most signifcant bits of addresses
&0F	1st file's sector address on disc
the format of last 8 bytes is repeated for remaining 30 file entries	

0 1	most significant 2 bits of total no. of sectors on disc
2	
3	
4 5	!BOOT start option as set by *OPT4,n
6	
7	

0 1	Top 2 bits of 1st file's start sector
2 3	Top 2 bits of 1st file's load address
4 5	Top 2 bits of 1st file's length in bytes
6 7	Top 2 bits of 1st file's sector address

16.3.4 The Advanced Disc Filing System

The ADFS is provided with the Master series computers and with the Plus 3 expansion unit for the Electron. The original BBC model B is unable to use ADFS unless a suitable 1770 upgrade has been fitted.This replaces the 8271 floppy disc controller chip, which is unable to use double density disc formats.

The ADFS implements the full filing system facilites as described in section 16.1.

The ADFS disc format

The ADFS uses tracks divided into 16 x 256 byte sectors. The sectors on a disc are given absolute sector numbers between 0 and the total number of sectors per disc. There are 640 sectors on a 40 track single sided disc, 1280 sectors on an 80 track single sided disc and 2560 sectors on a double sided 80 track disc.

The first two sectors of the disc are used to house the free space list as described in the following table.

Sector 0

&00 &02	start sector of 1st free space
&03 &05	start sector of 2nd free space
&06 &08	start sector of 3rd free space
&F3 &F5	start sector of 82nd free space
&F6 &FB	reserved
&FC &FE	total no. of sectors (LSB 1st)
&FF	checksum on sector 0

Sector 1

&00 &02	no. of sectors in 1st free space
&03 &05	no. of sectors in 2nd free space
&06 &08	no. of sectors in 3rd free space
&F3 &F5	no. of sectors in 82nd free spc.
&F6 &FA	reserved
&FB &FC	disc identifier
&FD	boot option number (*OPT4,n)
&FE	pointer to end of free space list
&FF	checksum on sector 1

The root directory is stored in the following 4 sectors of the disc. The format for any directory on the disc (root or sub-directory) is shown in the following table.

Sectors 2 - 6 (for root directory)

&00	BCD cycle number for directory
&01 &04	directory identifier string
&05 &0E	name and access string for 1st file in directory
&0F &12	load address for 1st file
&13 &16	execution address for 1st file
&17 &1A	length of 1st file (or sub-directory) in bytes
&1B &1D	start sector for 1st file (or sub-directory)
&1E	sequence number for 1st file
&1F &28	name and access string for 2nd file in directory
&29 &2C	load address for 2nd file
&2D &30	execution address for 2nd file
&31 &34	length of 2nd file (or sub-directory) in bytes
&35 &37	start sector for 2nd file (or sub-directory)
&38	sequence number for 2nd file
repeated for 47 directory entries	
&4C6	0
&4C7 &4D5	directory name & access string
&4D6 &4D8	start sector of parent directory
&4D9 &4E6	directory title
&4E7 &4F9	reserved
&4FA	BCD cycle number for directory
&4FB &4FE	directory identifier string
&4FF	0

16.3.5 The Network Filing System

Acorn's NFS software implements a full standard filing system with the following characteristics:

The second and third bytes of the attribute block are used to store the date when the file was created.

The device identity is not applicable, and its length is zero.

Directory and file names consist of 1 to 10 characters.

The media title is the disc title, and may be up to 16 characters long.

The NFS also implements a number of OSBYTE and OSWORD calls.

Very limited information about the Econet filing system is given here. For further information about this highly sophisticated system the reader should refer to the 'Econet User Guide' and the 'Econet Advanced User Guide'.

16.3.6 The Telesoft filing system

Files may be read from the teletext data broadcast by the television organisations as part of their teletext services.

Variations from the standard filing system implementation are:

OSFILE - only load operations are supported
OSARGS - only filing system identification supported
OSBPUT - not implemented
OSGBPB - not implemented
OSFIND - only input is supported (only 1 file)
OSFSC - calls 0 to 6 are implemented, call 3 returns a Bad command error.

File handles given are &14 or &15 for input only.

OSWORD call with A=&7A provides access to the various teletext commands from assembly language.

The Telesoftware ROM claims a total of 23 pages of absolute and private workspace.

16.3.7 IEEE filing system

The IEEE488 interface is a general purpose system for exchanging data between a number of devices in a local area. The control of transmission and reception of this information is handled by an IEEE filing system. This filing system behaves in the same way as other filing systems even

though the source or destination of the filing system data may not be a data storage device.

The IEEE filing system differs from the filing system standard in the following respects.

OSFILE - not supported
OSARGS - only filing system identification supported
OSGBPB - not implemented
OSFIND - only input and output (up to 16 channels.)
OSFSC - limited implementation

File handles given are in the range &F0 to &FF.

OSWORD call with A=&80 is used to invoke IEEEFS commands from assembler.

The IEEEFS paged ROM claims 2 pages of absolute workspace and 1 page of private workspace

16.4 Floppy Disc Hardware

Introduction

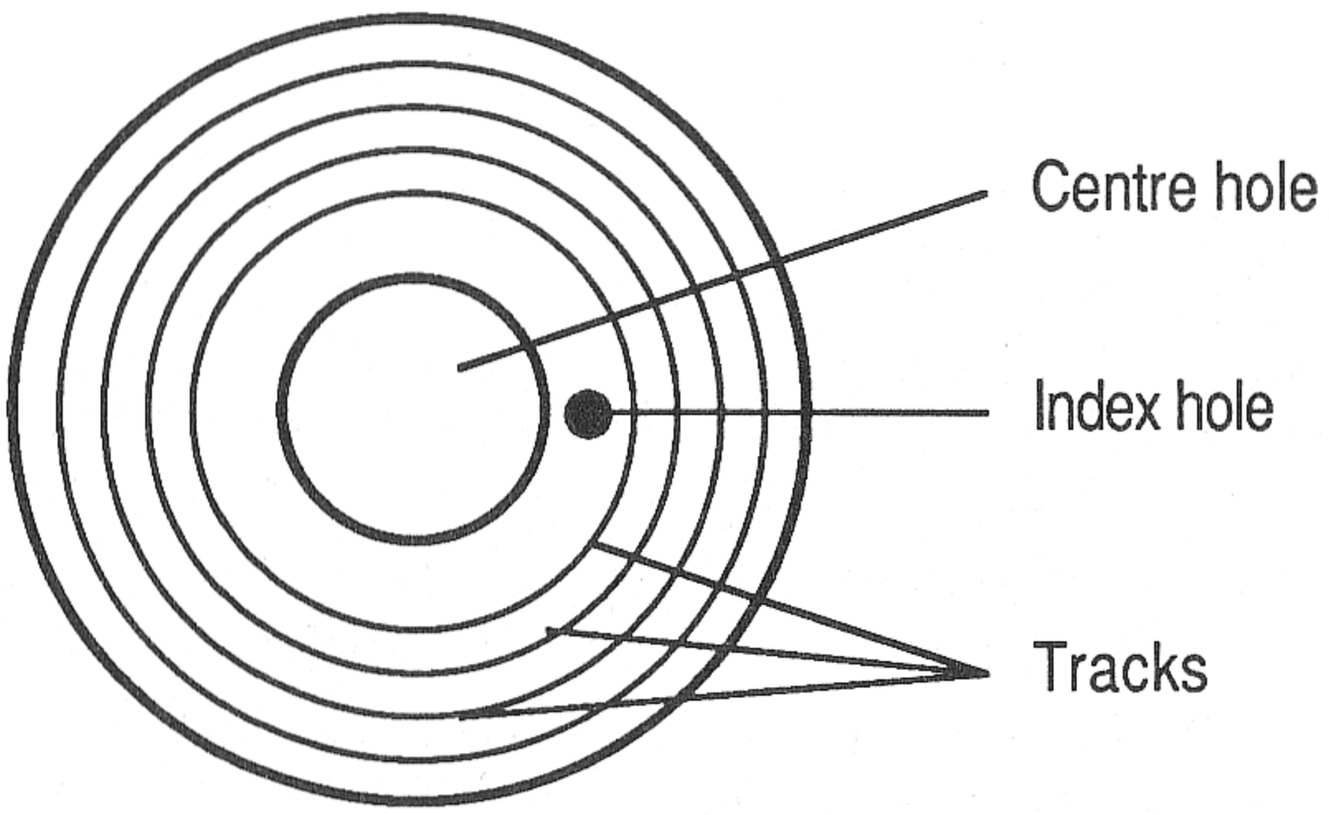
When the model B was first launched by Acorn most people used the cassette filing system and an attached audio tape recorder for storage and retrieval of programs. In the intervening years the price of floppy disc drives and control hardware has fallen dramatically. The stage has now been reached where practically all serious users have at least one floppy disc drive attached to their computer.

For everyday usage the typical user of the Acorn-BBC range will make all accesses to the floppy discs via the DFS or ADFS filing systems. However, the disc format employed by these filing systems (see the appropriate user manual for details) is not compatible with most other common computers like the IBM PC or CP/M machines. This normally means that data transfer between these machines can only be effected over a standard interface like the RS232 serial link. It would however sometimes be useful to be able to read and write discs from other types of computers. This would for example allow a colleague from overseas to send you a disc written on his IBM PC and containing his memoires! By directly controlling the floppy disc drive system you could actually read his memoires into your BBC Micro.

The remainder of this section describes how data is organised on a floppy disc and how you can achieve direct control of the disc hardware. This is followed by a simple example which reads a sector from a PC compatible diskette.

16.4.1 Floppy disc organisation

Each floppy disc consists of a double-sided flat disc coated in magnetically sensitive material. Data can be written to and read from this magnetic surface by a miniature read/write head. Single-sided drives have only one head and can therefore only access one surface of the floppy disc. Double-sided drives have two heads, one for each side of the disc. Near to the centre of the disc is a small hole. This is called the 'index hole'. A small light sensor is placed below the hole with a light emmiter above the hole. An 'index pulse' is therefore produce once per revolution of the disc, thereby defining the start position for reading or writing data. To allow the read/write head to access data stored at any position on the disc surface, it can step in towards the centre of the disc or out from the centre of the disc. Most drives have either 40 or 80 steps between the inner and outermost tracks on the disc. The head must be in close contact with the magnetic surface to read and write data (just like the head in a conventional cassette tape recorder). To allow the discs to be removed, the head is only loaded onto the disc surface briefly to allow read or write operations. It is unloaded or removed a short distance away from the surface when reading or writing has been finished.



The surface of a floppy disc

The diagram illustrates a typical disc surface. Note the index hole and the tracks. Each track is divided into manageable sector sizes typically 256 or 512 bytes long. To enable each sector to be identified each one has an associated ID field.

16.4.2 Floppy disc controller chips

To ensure that data can be written and read reliably, it is essential to ensure that all operations occur at precisely the correct instant. Each byte of data must be written consecutively as a stream of bits. Each sector must have a gap between it and the next sector to allow for minor fluctuations in the speed of disc revolution (all drives vary slightly). Extra checksum data must be written to detect when read data differs from the data which was written. Time must be allowed for the motor to spin the disc up to speed before attempting to read or write data. Because of the complex nature of these problems and more, the chip manufacturers produce a range of specialised controller chips. These interface to microprocessors like the 6502 and allow the programmer to issue a range of simple commands to position the heads and read or write data.

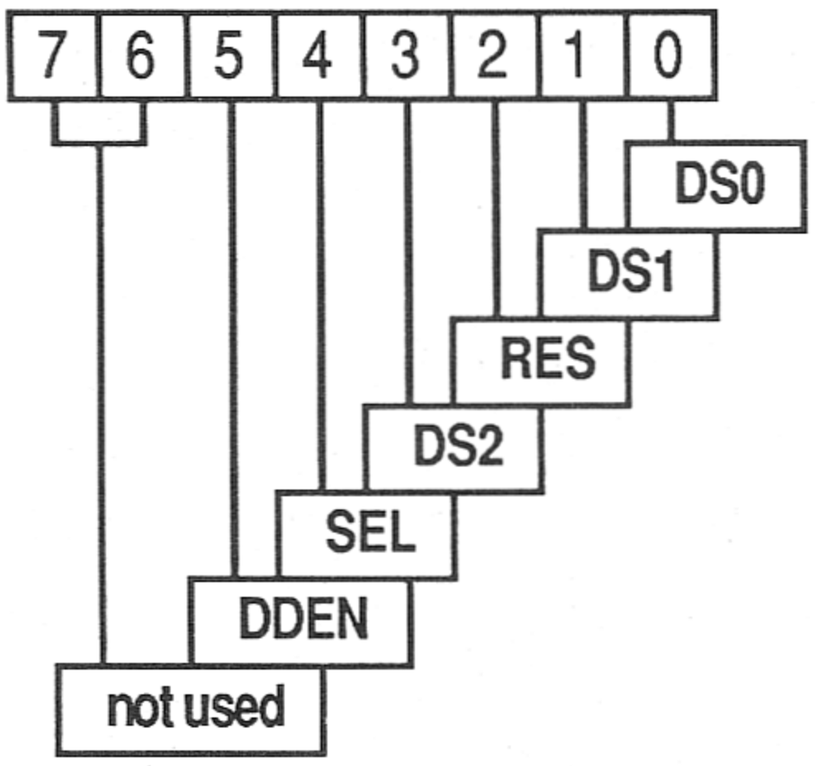
The model B used an early disc controller chip called the 8271. This was a very expensive chip which rapidly became obsolete. The model B+ and Master series use the WD1770 range of interface controller chips. It is this chip family which is covered in detail here.

1770 Control Registers

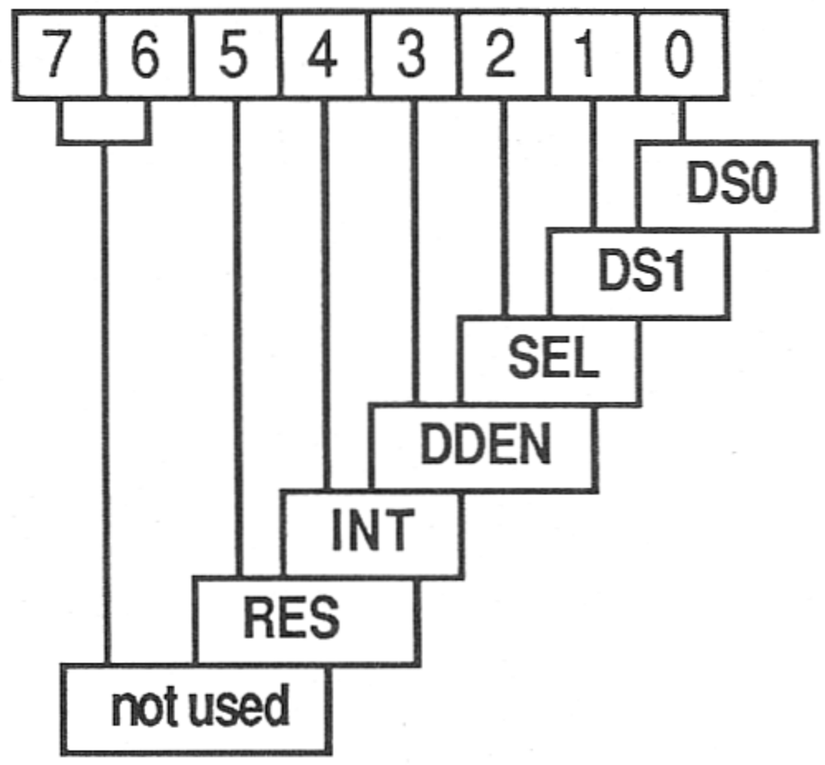
Model B+ address	Master address	Read function	Write function
&FE84	&FE28	Status register	Command register
&FE85	&FE29	Track register	Track register
&FE86	&FE2A	Sector register	Sector register
&FE87	&FE2B	Data register	Data register
&FE80	&FE24	none	Drive control register

The drive control register is actually a separate chip from the WD1770. It controls drive selection and various other functions as follows:

Master Drive Control Register
(&FE24)



B Plus Drive Control Register
(&FE80)



DDEN - this bit controls whether the 1770 is operating in single density (FM) or double density (MFM) mode. It must be set to '0' for double density or '1' for single density.

SEL - side select controls which side of the disc is accessed. When set to '1', side 1 is selected and when set to '0', side 0 is selected.

DS2, DS1, DS0 - are the drive select bits. When set to '1' the associated drive will be selected. Up to three drives can be attached to the Master and two drives can be attached to the B Plus. Only one drive should ever be selected at any one time, the other drive select bits must be set to '0'. Note that actual selection of the drive only occurs when the motor is turned on.

RES - this bit should be pulsed low for at least 50µs to reset the 1770 chip.

16.4.3 1770 operational overview

Before attempting to read or write data to a disc, the read/write head must be positioned at the correct location on the disc surface. The relevant disc drive is first selected using DS0, DS1 or DS2, the correct side is selected using SEL, and the correct recording density is selected using DDEN. The head can then be moved to the correct track. This is achieved using a 'type I' command including restore, seek, step-in and step-out. The next stage is to actually read or write data. This is achieved with either a 'type II' command which reads or writes a single sector, or a 'type III' command which reads or writes an entire track of data (multiple sectors).

Each command generates an NMI to the 6502 after completion. Data is also read or written by the 6502 in the NMI routine. Each byte read or

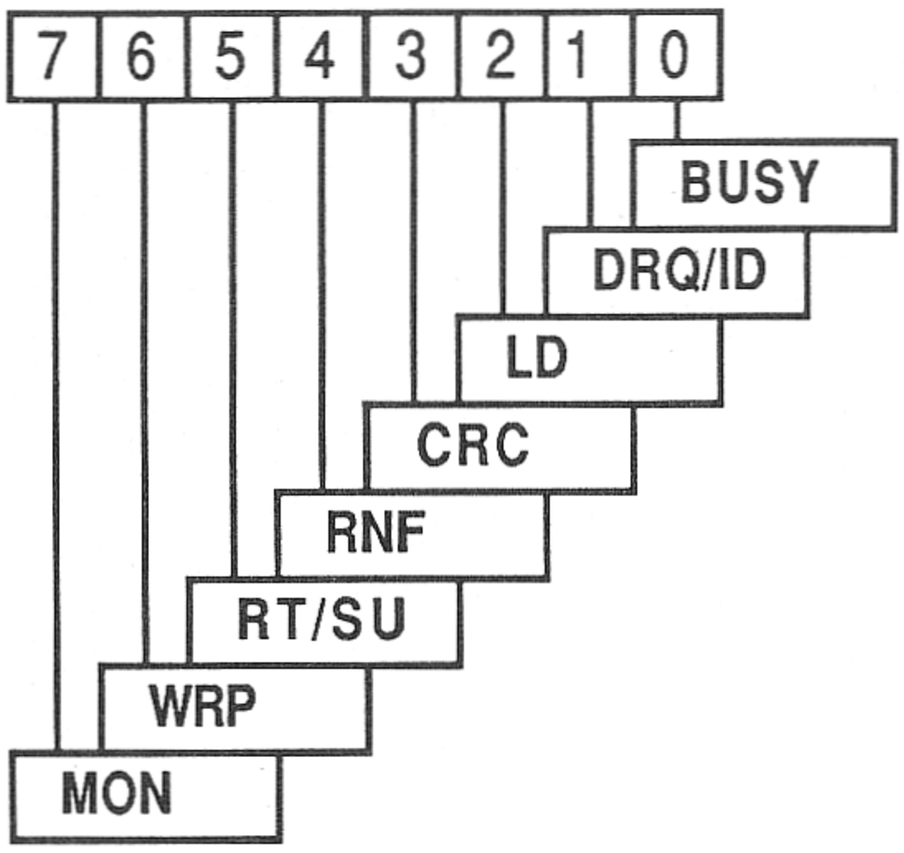
write has an associated NMI so the servicing of these interrupts needs to be exceptionally fast.

1770 command summary table

Type	Command	b7	b6	b5	b4	b3	b2	b1	b0
I	Restore	0	0	0	0	h	v	r1	r0
I	Seek	0	0	0	1	h	v	r1	r0
I	Step	0	0	1	u	h	v	r1	r0
I	Step-in	0	1	0	u	h	v	r1	r0
I	Step-out	0	1	1	u	h	v	r1	r0
II	Read sector	1	0	0	m	h	e	0	0
II	Write sector	1	0	1	m	h	e	p	a0
III	Read address	1	1	0	0	h	e	0	0
III	Read track	1	1	1	0	h	e	0	0
III	Write track	1	1	1	1	h	e	p	0
IV	Force interrupt	1	1	0	1	i3	i2	i1	i0

h = motor on control	h=0 enable disc spin up sequence h=1 disable disc spin up sequence
v = verify position	v=0 do not verify correct track v=1 verify on destination track
r1, r0 = stepping rates	r1 r0 WD1770 WD1772 0 0 6 ms 6 ms 0 1 12 ms 12 ms 1 0 20 ms 2 ms 1 1 30 ms 3 ms
u = update track register	u=0 do not update u=1 update track register
m = multiple sectors	m=0 do single sector operations m=1 do multiple sector operations
a0 = data address mark	a0=0 write normal data mark a0=1 write deleted data mark
e = 15ms settling delay	e=0 do not delay e=1 delay 15ms for head settling
p = write precompensation	p=0 enable write precompensation p=1 disable write precompensation
i3-i0 interrupt condition	i0=1 not used i1=1 not used i2=1 NMI on index pulse i3=1 immediate NMI i3,2,1,0=0 terminate without NMI

1770 Status Register - &FE28 Master (&FE84 B Plus)



MON - motor on signal =1 when the motor is turned on.

WRP - write protect, when set during write operation means that the disc is write protected

RT/SU - record type/spin-up. On type I commands means that the motor spin-up sequence has completed. On types II and III this bit indicates the record type 0 = data mark, 1 = deleted data mark.

RNF - record not found, when set means that the desired track, sector or side was not found. Reset when updated.

CRC - if set this bit indicates that a CRC (cyclic redundancy check) error was found in one or more ID fields or a data field.

LD - lost data/byte indicates that the 6502 did not respond to the NMI for data read within one byte time. On type I commands this bit is set to 1 except when the head is on track zero when it is set to 0.

DRQ/ID - data request/index. When set it indicates that the DR is full during read or empty during write. It should never be necessary to test this bit because an NMI is generated whenever data is ready or needed. On type I commands this bit reflects the status of the index hole sensor.

BUSY - when set indicates that a command is currently under execution. When reset no command is under execution. You should NOT read from the status register to check if a command is under execution, as this will reset the NMI if it occurs just at that moment.

1770 Type I commands

For all types of commands except type IV interrupt control the, h bit should be set to 1 to enable a delay of 6 index pulses (i.e. 6 complete revolutions of the disc) to allow the motor to reach its operating speed. If, after finishing a command, no more commands are issued within 9 revolutions, the motor automatically turns off. Consecutive read and/or write operations can be carried out once the motor is running at speed without waiting for the spin-up period. The r1 and r0 bits control the track to track stepping rate. The verify bit if set will ensure that the first encountered ID field is read and compared with the contents of the track register. If the track and ID field track number compare, and the CRC is correct, an NMI is generated with no errors. If the tracks match, but there is a CRC error, the CRC error status bit is set and the next encountered ID field is read to check for a match.

Restore (seek to track 0) - when issued by the 6502 instructs the 1770 to step until track 0 is reached. When reached, the track register is loaded with 0 and an NMI is generated. If track 0 is not reached within 255 steps (faulty drive) the 1770 stops stepping, generates an NMI and sets the 'seek error' status bit if v=1.

Seek - assumes that the track register holds the current track position (you must update this correctly if using multiple drives). Load the data register with the desired track location and issue a seek command. The head will be stepped to the relevant track, and verified against data there (if v=1). It will then be followed by an NMI.

Step - this command steps one track in the same direction as the previous stepping operation. If u=1 the track register is updated. A verification occurs if v=1. NMI occurs on completion.

Step-in - same as step, but steps towards innermost track.

Step-out - same as step, but steps towards outermost track.

1770 Type II commands

These are the read and write sector commands. When issued, the ID fields on the track are compared with the desired sector number until a match is found, at which point the data field is either written into or read from the data field. If no valid ID field is found within 5 revolutions of the disc, the record not found (RNF) bit in the status register is set and the command terminates with an NMI.

The m bit controls whether multiple sectors are to be read or written. If m=0, a single sector is accessed and an interrupt is generated at the completion of the command. If m=1, multiple sectors can be accessed and the sector register is internally updated so that the next

encountered ID field will be valid. The sector register is incremented by one after each sector read or write until the sector register exceeds the number of sectors on the track or until the force interrupt command is loaded into the command register.

Read sector - when this command is issued, the 1770 reads the ID fields on the current track until a sector match is found. The 'data address mark' (DAM) should then be found within 30 bytes (single density) or 43 bytes (double density) of the last ID field CRC byte. If the DAM is not found, the process is repeated for up to 5 revolutions, after which the RNF status bit is set and the command is terminated. When the first byte of data is ready to be read into the DR, an NMI is generated to the 6502. The byte must be read within 32µs (double density) or 64µs (single density) otherwise the DR will not be free to receive the next byte read from disc. If this occurs the last character is lost and the 'lost data' status bit is set. This sequence of reading bytes continues until the last byte in the sector has been read by the 6502. If a CRC error is detected at this point the command is terminated (even if it is a multiple sector read) and the CRC error status bit is set. At the end of the read operation the type of data address mark found in the data field is recorded in the status register.

Write sector - similar to read sector. Once an ID field with the correct track number, sector number, and the CRC is located, an NMI is generated. The 1770 counts 11 bytes (single density) or 22 bytes (double density) from the CRC field end. Assuming that the 6502 has loaded the first byte to be written into the DR, the 1770 writes 6 bytes of zeros (single density) or 12 bytes of zeros (double density) followed by the DAM as defined by a0. The data bytes are then written, an NMI being generated as soon as the next data byte can be loaded into the DR. If the byte is not loaded within 32µs (double density) or 64µs (single density) the 'lost data' status bit is set and a byte of zeros is written on the disc. The command is not terminated. A two byte CRC is computed by the 1770 and written after the last data byte. To write part of a sector, always write the whole sector with the unwanted bytes padded with zeros.

1770 Type III commands

Read ID field - the next 6 byte ID field encountered after issuing this command can be read by the 6502. An NMI is generated when each byte becomes available in the DR. The bytes read contain:

Byte 1 = track address
Byte 2 = side number
Byte 3 = sector address

Byte 4 = sector length
Byte 5 = CRC 1
Byte 6 = CRC 2

The 1770 still checks the CRC and sets the status bits accordingly. An NMI is generated after all 6 bytes have been read.

Read track - this command reads all gap, header and data bytes present on the track. It may prove useful for diagnostic purposes. The first byte is read immediately after detecting the index pulse. The 'busy' status bit is set whilst this command is in operation. Once completed, the 'busy' status bit is reset and a final NMI is generated.

Write track formatting the disc - this is normally performed only once when the disc is first formatted. A detailed discussion of the many different requirements for successful compilation of the necessary data is rather complicated and would take too much space to include here. The gory details can be found in the full manufacturer's data sheet on the WD1770.

1770 Type IV commands

To terminate a multiple sector read or write command, or to ensure type I status in the status register, a forced interrupt command can be issued. If a command is currently under execution, it will immediately be terminated and the busy status bit will be reset.

i3 = 1 causes and immediate interrupt
i2 = 1 causes an interrupt on every index pulse
i3,i2,i1,i0 ALL = 0 terminates current command without an NMI

An immediate interrupt is not automatically cleared by writing a new command or reading the status register. It must be cleared by issuing command &D0. Wait for at least 16µs (single density) or 32µs (double density) before writing a new command after issuing a forced interrupt.

16.4.4 Using the 1770 disc controller; an example program

The following example enables a file on a 360 Kb IBM format disk to be examined on an Acorn BBC micro with a 1770 disc controller.

The IBM disc format may be found in *The Peter Norton Programmer's Guide to the IBM PC* published by Microsoft Press. The IBM disc format relies on a directory containing the filename etc. and a file allocation table (FAT) which contains information about the allocation of clusters

(groups of disc sectors) to each file. The FAT contains a series of 12 bit values, one for each cluster, containing the cluster number of the next cluster in that particular data chain. This limited information may help the reader follow this program but unfortunately there is not room for a more detailed description here.

```
  1 REM IBM disc dump utility program by Mark Holmes

  2 machine$="MASTER": REM if using B+ replace string with "B+"

 10 REM Main program loop
 20 MODE7
 30 DIM buffer &1500
 40 buffer=(buffer+&FF) AND &FF00
 50 PROCsetup
 60 REPEAT
 70   PROCprompt("Press D(ir),F(ile dump) or Q(uit)")
 80   K%=GET
 90   IF K%=ASC"D" OR K%=ASC"d" PROCdisplay_directory
110   IF K%=ASC"F" OR K%=ASC"f" PROCfile_dump
120 UNTIL K%=ASC"Q" OR K%=ASC"q"
130 MODE7
140 END

200 REM Print out IBM disc directory
210 DEF PROCdisplay_directory
220 CLS
230 cluster=FNget_dir("display directory")
240 ENDPROC

300 REM Dump contents of file to screen
310 DEF PROCfile_dump
320 VDU 28,3,24,39,24,12
330 INPUT TAB(4);"Enter Filename",name$
340 VDU 28,0,23,39,2
350 PROCprompt(name$)
360 CLS
370 cluster=FNget_dir(FNname_convert(name$))
380 IF cluster=-1 PRINT TAB(10,5);name$;" not found":
                  PROCprompt("Press any key to continue"):
                  K%=GET:ENDPROC
390 IF cluster=0  PRINT TAB(10,5);name$;" has no data":
                  PROCprompt("Press any key to continue"):
                  K%=GET:ENDPROC
400 block_address%=0
410 REPEAT
420   PROCconvert(cluster)
430   cluster=FNnext_cluster(cluster)
440   PROCdump(buffer,&200,block_address%)
450   block_address%=block_address%+&200
460 UNTIL cluster>&FEF
470 ENDPROC

500 REM Loop until WD1770 is not busy
510 DEF PROCwait
520 REPEAT:UNTIL((?status_reg) AND busy_bit)<>1
530 ENDPROC

600 REM Read sector from disc
610 DEF PROCread_sector(side,track,sector,address)
620 IF address<>0 ?(save+1)=address MOD 256: ?(save+2)=address DIV 256
630 IF side=1 value=(value OR &10) ELSE value=(value AND &EF)
640 ?dr_ctrl_reg=value
650 ?data_reg=track
660 ?comm_reg=&19
```

```
670 PROCwait
680 ?track_reg=track DIV 2
690 ?sector_reg=sector
700 ?comm_reg=&84
710 PROCwait
720 IF ((?status_reg) AND error_bit)=&10
                          PRINT"DATA ERROR",side,track,sector:STOP
730 ?track_reg=track
740 ENDPROC

800 REM Initialise disc drive control register
810 DEF PROCdisc_init(drive,side,density)
820 value=4
830 IF drive=1 value=value+2 ELSE value=value+1
840 IF side=1 value=value+&10
 850 IF density<>2 value=value+&20
 960 ?dr_ctrl_reg=value
 870 ?comm_reg=&09
 880 PROCwait
 890 ENDPROC

 900 REM Search for filename in directory
 910 DEF FNget_dir(search$)
 920 I=dir_buffer
 930 REPEAT
 940   IF ?I<>&E5 B=FNshow_name
 950   I=I+32
 960 UNTIL (I=dir_buffer+112*32) OR (?I=0) OR B<>-1
 970 =B

1000 REM Compare search name to directory entry,
1001 REM return cluster number or -1 if no file.or 0 if empty file
1010 DEF FNshow_name
1020 fname$=""
1030 FOR J=0 TO 10
1040   fname$=fname$+CHR$(I?J)
1050 NEXT
1060 fat=(I!26) AND &FFFF
1070 IF search$="display directory" PRINTLEFT$(fname$,8);".";
                                              RIGHT$(fname$,3);"  ";
1080 IF POS=2 VDU 13
1090 IF fname$=search$ =fat ELSE =-1

2000 REM Return next cluster from the file allocation table
2010 DEF FNnext_cluster(start)
2020 fat_entry=fat_buffer!((start DIV 2)*3)
2030 byte1=fat_entry AND &FF
2040 byte2=(fat_entry AND &FF00)/&100
2050 byte3=(fat_entry AND &FF0000)/&10000
2060 IF start MOD 2=0 =byte1+((byte2 AND &F)*&100)
2070 IF start MOD 2=1 =byte2/&10+(byte3*&10)

3000 REM convert cluster number to physical disc location
3010 DEF PROCconvert(cluster_no)
3020 log_sector=(cluster_no-2)*2+12
3030 phys_side=(log_sector DIV 9) MOD 2
3040 phys_track=log_sector DIV (9 * 2)
3050 phys_sector=1+log_sector MOD 9
3060 PROCread_sector(phys_side,phys_track*2,phys_sector,buffer)
3070 ENDPROC

4000 REM dump data to screen
4010 DEF PROCdump(where%,how_much%,file_offset%)
4020 FOR I%=where% TO where%+how_much%-8 STEP8
4030   @%=6
4040   PRINT CHR$(129);~file_offset%+I%-where%,CHR$(135);
4050   @%=1
```

```
4060    FOR J%=0 TO 7
4070      PRINT~(I%?J% AND &F0)/&10,~I%?J% AND &F," ";
4080    NEXT
4090    PRINT CHR$(8);CHR$(131);
4100    FOR J%=0 TO 7
4110      IF (I%?J%>31) AND (I%?J%<127) PRINTCHR$(I%?J%);
                                        ELSE PRINT".";
4120    NEXT
4130    PRINT
4140 NEXT
4150 ENDPROC

5000 REM setup variables, load directory and FAT etc.
5010 DEF PROCsetup
5020 IF machine$="MASTER" dr_ctrl_reg=&FE24 ELSE dr_ctrl_reg=&FE80
5030 IF machine$="MASTER" WD1770_addr=&FE28 ELSE WD1770_addr=&FE84
5040 comm_reg=WD1770_addr
5050 status_reg=WD1770_addr
5060 track_reg=WD1770_addr+1
5070 sector_reg=WD1770_addr+2
5080 data_reg=WD1770_addr+3
5090 busy_bit=&01
5100 error_bit=&10

5110 FOR opt%=0 TO 3 STEP 3
5120  P%=&0D00: REM NMI address
5130  VDU21
5140  [
5150        OPT opt%
5160        PHA                 \ save accumulator on stack
5170        LDA status_reg      \ get status register value
5180        AND #&1F            \ mask out unwanted bits
5190        CMP #&03            \ data ready & busy bits
5200        BNE exit            \ not interested in this NMI
5210        LDA data_reg        \ get data from data register
5220  .save STA &2000           \ store data
5230        INC save+1          \ increment save address LSB
5240        BNE &0D18           \ not page boundary so exit
5250        INC save+2          \ increment save address MSB
5260  .exit PLA                 \ restore accumulator
5270        RTI                 \ return from interrupt
5320  ]
5330  NEXT opt%
5340 VDU6
5350 VDU23,1;0;0;0;0
5360 PRINT CHR$(135);CHR$(157);CHR$(132);CHR$(141);
5400 PRINT TAB(8);"IBM disc dump utility"
5410 PRINT CHR$(135);CHR$(157);CHR$(132);CHR$(141);
5420 PRINT TAB(8);"IBM disc dump utility"
5430 PRINT TAB(0,24);CHR$(135);CHR$(157);CHR$(132);
5440 PRINT TAB(0,23);CHR$(135);CHR$(157);CHR$(132);
5450 PROCprompt("Press any key to continue")
5460 VDU 28,0,23,39,2,12
5470 PRINT TAB(5,5);"Put IBM format disc in drive 0"
5480 K%=GET
5490 CLS
5500 fat_buffer=buffer+&200
5510 dir_buffer=buffer+&200+&400
5520 drive=0:density=2:side=0
5530 PROCdisc_init(drive,side,density)
5540 PROCread_sector(0,0,2,fat_buffer)
5550 PROCread_sector(0,0,3,0)
5560 PROCread_sector(0,0,4,0)
5570 PROCread_sector(0,0,5,0)
5580 PROCread_sector(0,0,6,dir_buffer)
5590 PROCread_sector(0,0,7,0)
5600 PROCread_sector(0,0,8,0)
```

```
5610 PROCread_sector(0,0,9,0)
5620 PROCread_sector(1,0,1,0)
5630 PROCread_sector(1,0,2,0)
5640 PROCread_sector(1,0,3,0)
5650 ENDPROC

6000 REM Place message at bottom of screen
6010 DEF PROCprompt(message$)
6020 VDU 28,3,24,39,24,12
6030 PRINT TAB((34-LEN(message$))/2,0);message$;
6040 VDU 28,0,23,39,2
6050 ENDPROC

7000 REM Convert filename into directory type format
7010 DEF FNname_convert(string_in$)
7020 string_out$=""
7030 I%=1
7040 REPEAT
7050   string_out$=string_out$+MID$(string_in$,I%,1)
7060   I%=I%+1
7070 UNTILMID$(string_in$,I%,1)="." OR I%=LEN(string_in$)+1
7080 IF LEN(string_out$)<8 REPEAT string_out$=string_out$+" ":
                           UNTIL LEN(string_out$)=8
7090 IF MID$(string_in$,I%,1)="." I%=I%+1
7100 IF I%<=LEN(string_in$) REPEAT
                            string_out$=string_out$+MID$(string_in$,I%,1):
                            I%=I%+1:
                            UNTIL I%=LEN(string_in$)+1
7110 IF LEN(string_out$)<11 REPEAT
                            string_out$=string_out$+" ":
                            UNTIL LEN(string_out$)=11
7120 =string_out$
```

17 Paged ROMs

The Acorn BBC microcomputers allow a number of ROM based programs to be resident in a machine in the same address space. Each ROM is selected (paged in) as required and then de-selected (paged out) as software in another ROM is required. The B+ and Master series BBC microcomputers contain additional RAM which may be used in the same way as paged ROMS. This memory is described in chapter 12 and the description of paged ROMS below is equally applicable to this sideways RAM.

Paged ROMs work broadly in one of two ways. They act either as languages such as BASIC and LISP or as utilities such as filing systems and device drivers. Languages may also include such things as word processors and CAD graphics packages.

At any one time only one language should be active. Thus most machines will enter BASIC as the default language. The current language has access to, and control over the user RAM, which it can allocate to users e.g. for BASIC programs or word processing text.

While the one language is active any other ROM offering a service may be called upon as is appropriate. When a request for a service is generated the operating system interrogates each paged ROM in turn until the request is acknowledged and acted upon. Different types of request are indicated to each ROM by the operating system entering the service entry point of that ROM with an accumulator value representing the reason. These calls are called paged ROM service calls. If the service entry point is entered with A=7 this indicates that someone has asked the operating system for an OSBYTE call which the operating system failed to recognise and so is now asking the paged ROMs if they claim it. If a service call is recognised then the ROM should act upon it and clear the accumulator before returning control back to the operating system. If the ROM does not wish to claim the call it should return control to the operating system with the accumulator value unchanged.

There are two sets of paged ROMs, service ROMs and language ROMs. All language ROMs should respond to paged ROM service calls and so should be service ROMs as well. BASIC is an exception to this rule and the operating system recognises it by virtue of the fact that it is a language ROM not offering a service entry.

17.1 Paged ROM header format

In order to enable the operating system to recognise ROM types and treat them accordingly a protocol has been drawn up for a standard ROM format.

ROM offset	size	description
0	3	language entry (JMP address)
3	3	service entry (JMP address)
6	1	ROM type flag
7	1	copyright string offset pointer (=10+t+v)
8	1	version number (binary)
9	[t]	title string
9+t	1	zero byte
10+t	[v]	version string
10+t+v	1	zero byte
11+t+v	[c]	copyright string
11+t+v+c	1	zero byte
12+t+v+c	4	2nd processor relocation address
16+t+v+c	....	.. .. rest of ROM, code and data

Below is a full description of each field of the paged ROM format.

17.1.1 Language Entry

This should consist of a three byte JMP instruction referring to the language entry point. This code is called upon when a language is initialised. When a Tube is active, the language may be copied across to the second processor and then entered. When a language is copied across the tube it may be relocated to a different address (see section 17.4).

If a ROM is not a language ROM this field should contain zeros.

17.1.2 Service Entry

This should consist of a three byte JMP instruction referring to the service entry point. All paged ROM service calls are passed to each ROM service ROM present in a machine via this entry point (see section 17.4.1). All paged ROMS should have a service entry

17.1.3 ROM Type Byte

The value of this byte gives information to the operating system about the nature of the ROM.

The first 4 bits indicate the processor type for which the code is intended. This is of importance to second processors who may get languages copied across to them. A second processor will look for the correct value of these bits before attempting to run the language. The following values have been assigned:

bit values				dec. value	processor type
3	2	1	0		
0	0	0	0	0	6502 BASIC
0	0	0	1	1	reserved
0	0	1	0	2	6502 code (not BASIC)
0	0	1	1	3	6800 code
1	0	0	0	8	Z80 code
1	0	0	1	9	32016 code
1	0	1	0	10	reserved
1	0	1	1	11	80186 code
1	1	0	0	12	80286 code
1	1	0	1	13	reserved

Bit 4 of the ROM type flag is not used on machines other than the Electron where it indicates the presence of firm key expansions (see section 17.3.2).

If bit 5 is set, this indicates that the language code in this ROM has a relocation address on a second processor. This normally means that the machine code contained in the ROM can only run in the second processor when it is located at that address. Service routines are not executed from the Tube copy.

If bit 6 is not set, this indicates that the ROM is not a language and will not be considered for initialisation following a hard reset. This does not mean that the ROM cannot have a language entry point. However, if the language is entered via a service call (i.e. *<name>) a soft reset will reinitialise that language. Master series computers are more rigorous in their interpretation of this bit and will not allow any type of language entry if this bit is not set.

Bit 7 is used to indicate that a ROM has a service entry. All ROMs should have this bit set.

17.1.4 Copyright Offset Pointer

This is an offset value from the beginning of the ROM to the copyright string. It is important that this points to a string starting with the character values &28, &43 and &29 e.g. "(c)Adder 1987". The operating systems use this data to determine whether a ROM physically exists in a ROM position.

17.1.5 Binary Version Number

This eight bit version number of the software contained in a ROM helps identify software. This byte is not used by any operating system and need not correspond to the version string.

17.1.6 Title String

This is a string which is printed out as the operating system enters the ROM as a language.

17.1.7 Version String

This should be a string identifying the release number of the software. The format of this string should be A.BB where A and B are ASCII characters of decimal digits.

On entering a language the error pointer is set to the start of this string. If there is no version string, the error pointer is directed to the copyright string.

17.1.8 Copyright String

This string is essential for the operating system recognition of a paged ROM (see section 17.1.4 above). The copyright string should always be preceded by a zero byte and start with the characters '(c)'.

17.1.9 The Tube Relocation address

This is the address which is used when a ROM is relocated during copying across the Tube to a second processor.

The language code should be assembled to run at this address, but the service code should be assembled to run from &8000 as it will be executed within the ROM in the I/O processor.

Executing Software in Paged ROMs

It is possible to execute machine code in a paged ROM in one of three ways; via the language entry point after a reset, via the service entry point when the operating system performs a service call or via an extended vector (which is usually set up by a paged ROM in response to a service call).

17.2 Paged ROM/RAM installation

In the BBC microcomputer there were originally only 5 ROM sockets but with the addition of expansion hardware up to 16 paged ROMs may be made available. The BBC Master microcomputer is supplied with the equivalent of 7 paged ROMs already in the computer. A total of 64 K of sideways RAM is available for use as another 4 paged ROMs and the two ROM cartridges can hold another 4 paged ROMs between them. Three further ROM sockets may be found on the circuit board but this does not mean that the Master can use more than a total of 16 paged ROMs. Only one of the empty, on-board, sockets may be used without interfering with use of the sideways RAM. Use of each of the other two sockets leads to the loss of two 16k banks of sideways RAM.

The hardware for selecting ROMs was covered in section 12.3. Here we consider the practical requirements which are necessary to instsall paged ROMs and select sideways RAM.

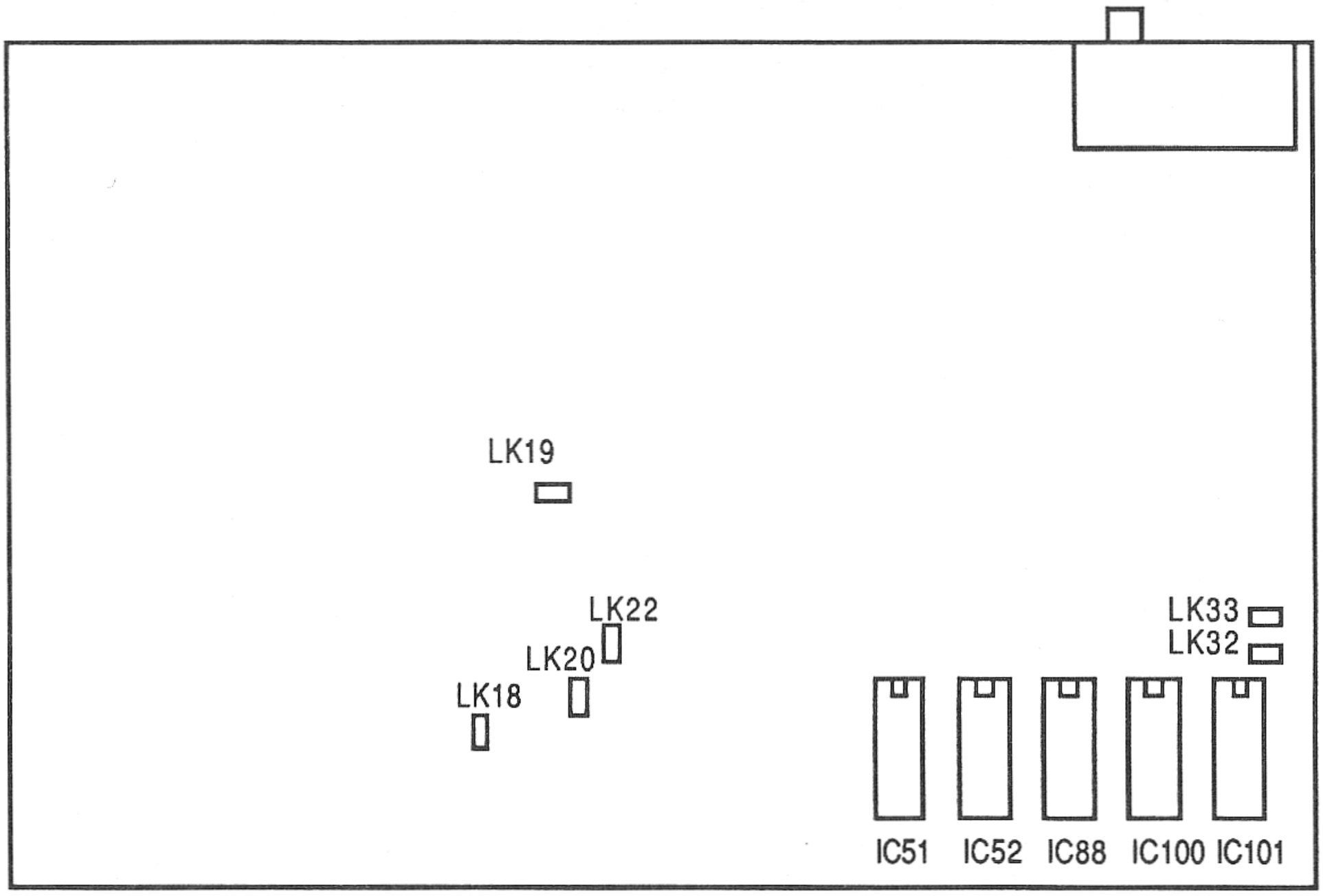
It is useful to briefly consider the various chips which may be plugged into the Acorn-BBC range of machines. Typically, commercially available software will be shipped in EPROMs (Prgrammable Read Only Memories). These are easily recognisable because of the window on top which allows ultra violet light to illuminate the chip surface directly and consequently erase it. These chips are initially erased or 'empty' with all locations containing &FF. A programmer (many are available for the BBC Micro) writes data into the chip, where it remains even when the power is disconnected. The following EPROM types are commonly used:

EPROM	size	microcomputer
2764	8K x 8	model B only
27128	16K x 8	model B, B Plus & Master
27256	32K x 8	model B, B Plus & Master
27513	16K x 4 x 8	Master only (ROM cartridge)

Each paged ROM can contain up to 16K bytes of data. Some ROM software doesn't occupy the full 16K and may fit into an 8K chip. When the model B was designed, the largest available EPROMs at acceptably low prices were the 27128s. As the prices for larger chips have fallen, the on board sockets have been changed to accept the larger capacity chips.

When installing extra ROMs remember that ROM number 15 is the highest priority ROM, whilst ROM 0 is the lowest priority.

17.2.1 Installing ROMs in the model B



ROM socket and link positions for model B

The model B has four paged ROM sockets (IC52, IC88, IC100 & IC101) on the main board, located in the lower right hand corner. The 16K BASIC ROM is normally plugged into IC52 and the DFS ROM (if fitted) is usually in IC88. All four sockets can be configured with hardware links to accept 8K/16K ROMs (or EPROMs). For all combinations described here, the following links should be set:

LINK 20 NORTH
LINK 22 NORTH

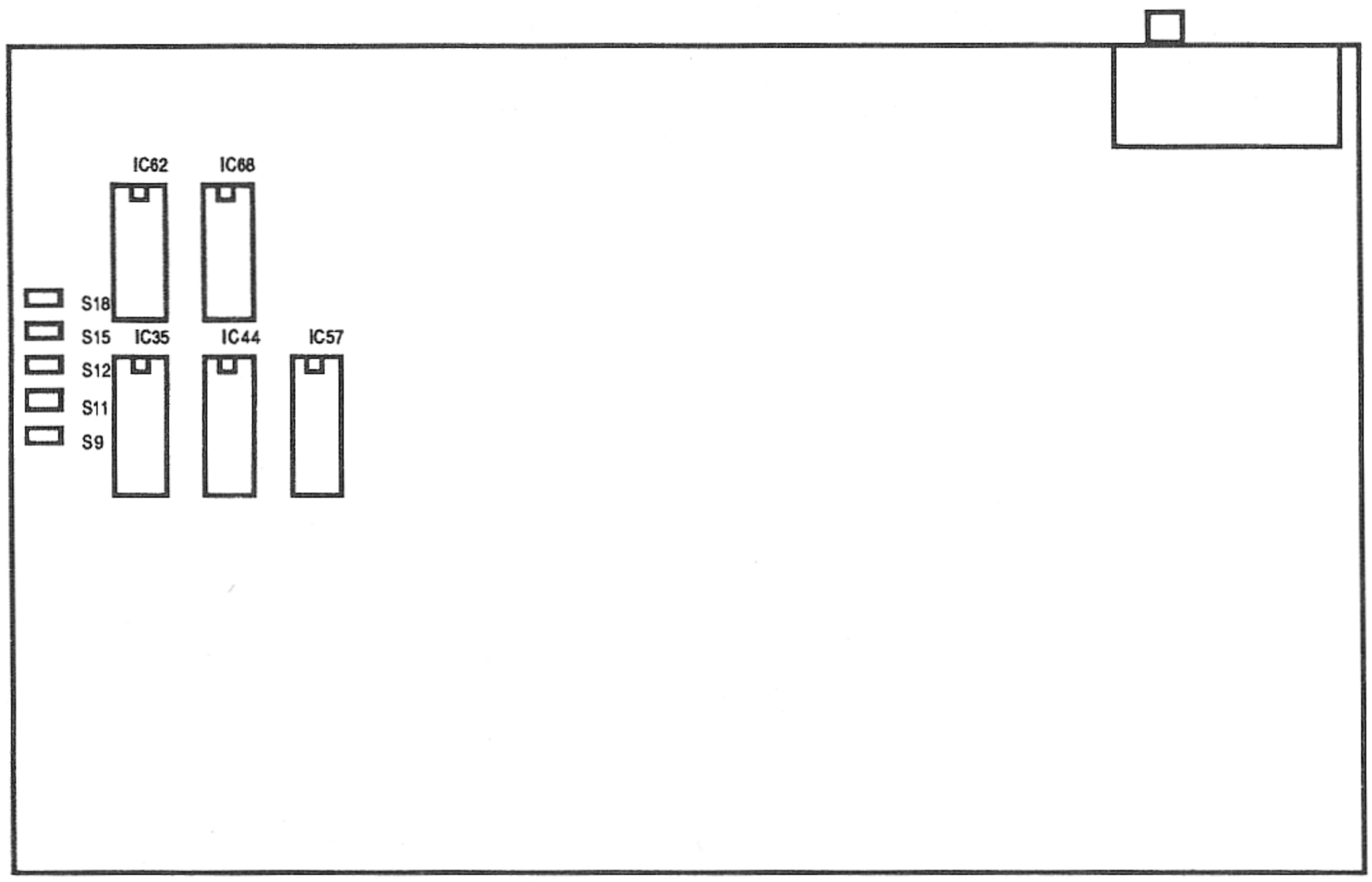
Socket	ROM ID	Link selections 16K	Link selections 8K
IC52 IC88	12 13	S32 WEST	S32 EAST
IC100 IC101	14 15	S33 WEST	S33 EAST

Note: Some old chips have longer access times than 250ns. To allow these chips to work reliably, the access speed can be slowed down by links 18 and 19 as follows:

Socket	Slow access	Fast access
IC100	Link 18 SOUTH	Link 18 NORTH
IC52, 88, 100	Link 19 WEST	Link 19 EAST

Always use fast access chips if possible.

17.2.2 Installing ROMs in the B Plus



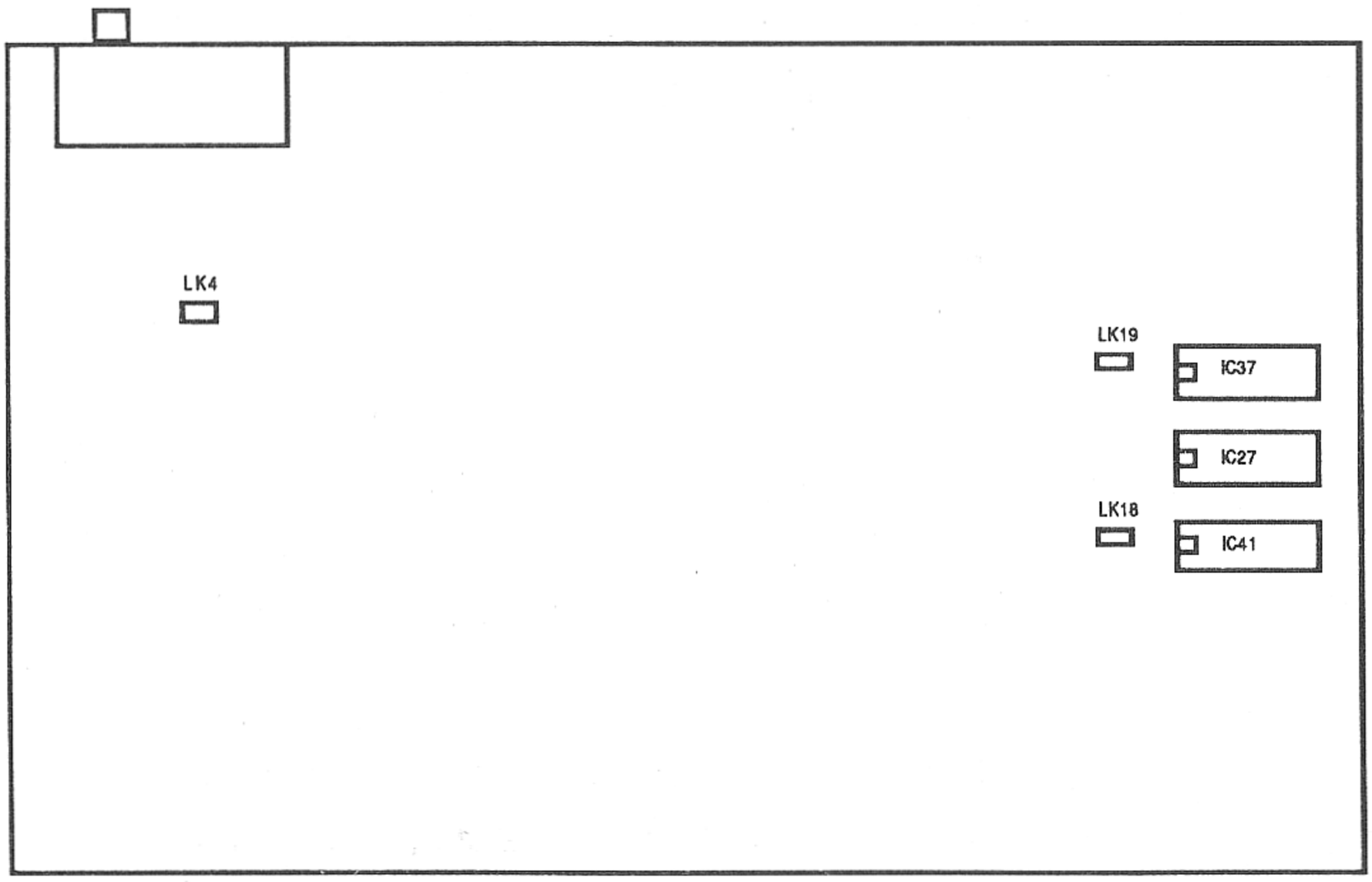
ROM socket and link positions for model B Plus

The B Plus has 5 ROM sockets available to the user, located in the top left-hand corner of the board. These sockets can each accept either a 16K or a 32K EPROM depending upon the setting of the associated links. If 32K EPROMs are used, two 'logical' paged ROMs can be stored in each physical chip.

Socket	ROM ID	Link (West = 16K , East = 32K)
IC68	10/11	S18
IC62	8/9	S15
IC57	6/7	S9
IC44	4/5	S11
IC35	2/3	S12

The MOS and BASIC are both contained in a 32K byte ROM. BASIC occupies ROMs 15/14.

17.2.3 Installing ROMs in the Master 128



ROM socket and link positions for Master 128

Unlike the earlier machines, the Master comes equipped with a massive 1Megabit ROM. This provides 128K bytes of ROM. 16K is allocated to the MOS at &C000-&FFFF. The remaining 112K is allocated as:

ROM ID	Contains
15	MOS extras, including the cassette filing system
14	VIEW - Acorn wordprocessor
13	Advanced Disc Filing System (ADFS)
12	BASIC language
11	EDIT - Acorn screen editor
10	Viewsheet - Acorn spreadsheet
9	Disc Filing System (DFS)

ROM number 8 can be plugged directly into the vacant 16K socket (IC27).

Master computers are shipped with four internal banks of 16K sideways RAM selected in ROM positions 4, 5, 6 and 7. These can be conveniently loaded with ROM software from disc. Users who wish to plug in more ROMs on the main board have to sacrifice the use of at least two of these sideways RAMs. ICs 41 and 37 can each accept a 32K physical ROM which may contain two 'logical' ROMs numbers 4/5 and 6/7 respectively.

ROM ID	Link selecting RAM	Link selecting	ROM socket
6/7	LK19 WEST	LK19 EAST	IC37
4/5	LK18 WEST	LK18 EAST	IC41

The remaining four ROM/RAM slots 0,1,2,3 address a cartridge plugged into the socket on the keyboard. One cartridge contains ROMs 0,1 and the other contains 2,3. To achieve extra ROM capacity, two 27513 chips can be used in each cartridge. More details are available in section 23.7 which describes the cartridge hardware.

17.2.4 Installing ROMs in the Master Compact

The Master Compact is supplied with a 128K system ROM. As in the Master, 16K is allocated to the MOS from &C000 to &FFFF and the remaining 112K is allocated as 7 x 16K paged ROMs numbers 9-15. Four 16K paged RAMs are located as paged ROMs numbers 4,5,6 and 7.

In addition to the system ROM, there are four sockets on the Compact's main circuit board. Three of these are hard-wired to accept only 16K EPROMs. The fourth can accept either a 16K or a 32K EPROM.

Socket	ROM ID	Type
IC38	0/1	16K/32K
IC23	2	16K
IC17	3	16K
IC29	8	16K

Since the ROM select to the cartridge interface also selects ROMs 0/1, it is necessary to select the correct position of these ROMs using link PL11. Setting this link SOUTH enables ROMs on the internal socket IC38 whilst setting the link NORTH enables ROMs in the cartridge ROM module.

17.3 Language ROMs

The term language ROM is something of a misnomer given that language ROMs need not contain languages. In the context of paged

ROM software, the language is the paged ROM that is in overall control. Other paged ROMs may perform functions transiently but control is then returned to the current language ROM. The language ROM receives a large allocation of zero page workspace and is allocated pages 4 through to 8 as private workspace. In addition to this the language has control of the user RAM which may or may not be used as additional workspace. BASIC, for example, uses a variable portion of the user RAM (from LOMEM to HIMEM) for the storage of program variables.

Languages are most typically implemented in language ROMs as would be expected. BASIC, FORTH, LISP and BCPL are all language ROMs. Other software implemented as language ROMs include word processors and terminal emulators.

No paged ROM software should be executed unless a service call has been performed first. The language entered after a hard reset will be the language ROM identified by the ROM type byte in its header occupying the highest priority socket. Following a soft reset the language active when the reset occurred will be reinitialised. The service entry code of a language ROM should recognise an unknown *command to allow entry of the language via the command line interpreter.

17.3.1 Language initialisation

When the language is selected a language ROM will be entered via the language entry point with an accumulator value of &01. The selected language is entered with a JMP instruction and no return is expected. The stack pointer should be reinitialised as the stack state is undefined on entry.

The carry flag is clear if the language has been entered from a reset (BREAK) and is set otherwise. The type of reset can be checked with OSBYTE &FD.

The language ROM should also be able to respond to service calls which may affect it (see section 17.4.1) e.g. be able to respond to the service call which warns of a changing OSHWM due to font explosion.

17.3.2 Electron firm keys

On the Electron the function keys are implemented as a combination key press requiring the use of the CAPS LK/FUNC key with the number keys. In addition to these soft keys there are a number of non-programmable firm keys which also produce text strings when pressed.

The other character keys (A to Z plus the comma, full stop and slash keys) pressed in combination with the CAPS LK/FUNC key constitute the firm keys.

A language ROM indicates that it has the facility to expand these keys by bit 4 of the ROM type byte being set (see section 17.1.3).

When the operating system detects that a firm key has been pressed it calls the language via its entry point to request the expansion of the key. The language should then yield the firm key string one character at a time in response to further calls.

The two calls made through the language entry point are:

A=2 This call expects the next key in the firm key expansion to be returned in Y.

A=3, Y=firm key code This call is an initialising call. The language should return the length of the firm key string in Y.

The key values passed to the language with this call are:

&80 to &89	FUNC+0 to FUNC+9
&90 to &A9	FUNC+A to FUNC+Z
&AA	FUNC+:
&AB	FUNC+;
&AC	FUNC+,
&AD	FUNC+=
&AE	FUNC+.
&AF	FUNC+/

The operating system inserts these key values into the input buffer as they are received.

OSBYTE &CC (204) may be used to read or write the OS copy of its firm key pointer and OSBYTE &CD (205) may be used to read or write the length of the current firm key string being expanded.

17.3.3 Memory allocated for language ROM use

Four pages of workspace are allocated for use by the currently selected language and 144 bytes of zero page in addition to main memory. Zero page workspace is required so that zero page addressing modes can be used and the language workspace is important because of its fixed position in the memory map in comparison to main memory whose boundaries are variable.

OSHWM to HIMEM	main memory (boundaries variable)
&400 to &7FF	language workspace
&00 to &8F	language zero page workspace

These areas are reserved for the use for languages running on non-Tube machines and on 6502 second processors.

17.3.4 Language ROM compatibility

The first question that a programmer should consider before implementing software in a Language type ROM is whether it actually needs to be a language ROM? Many utilities are only required transiently and it is better to implement them as service type ROMs. A routine in a service type ROM can then be used from the language environment.

The language should have a service entry point so that it may be selected by a *command and be able to respond to changes in OSHWM. It must be remembered however that a 6502 language ROM is copied across to a 6502 second processor when the Tube is active. When running on a second processor, service calls will not be received by the language ROM. The service code should not use the language work space (&400-&7FF or language zero page) because the service code is executed in the i/o processorand has the equivalent status of the current 'language'. The language code should not attempt to perform any manipulation of hardware by direct poking as this would make it machine dependent. The programmer may wish to implement hardware dependent routines in the service section of the ROM. The language code should communicate with the service code using unknown OSBYTE calls etc. for this purpose.

17.4 Service ROMs

Service ROMs contain code which is entered via the service entry point. Service ROM code will always be executed in the ROM itself i.e. always in the i/o processor c.f. language ROMs. The calls made by the operating system to service ROMs are called paged ROM service calls but will usually be referred to as just 'service calls'.

The type of software which might be implemented in service type ROMs are filing systems, user printer drivers, extension VDU commands and languages; In fact just about anything. It should be noted that extreme care should be taken to implement paged ROM service routines correctly.

For example let us take a graphics extension ROM. This would be a service ROM because it has to be initialised by using the unknown *command service call (or possibly by the boot program call). However, the actual extension to the graphics commands would be performed by intercepting the VDU extension vector. This would be directed at the ROM code using an extended vector (see section 17.4.3). It would be foolish to call routines by the unknown *command mechanism which then returned values to BASIC by poking into BASIC's single letter integer variable storage space. This is because when a Tube is active the graphics routines would corrupt the Tube routines in the language workspace. It is also foolish not to take advantage of the elegant expansion capabilities which are provided.

To understand how software can be incorporated into a paged ROM, be interfaced correctly with the operating system and thus executed at the appropriate time, an understanding of paged ROM service calls is essential.

When a hard reset occurs the operating system makes a note of which paged ROM sockets contain valid ROMs. Subsequently, as the machine carries out its various tasks, service calls and language requests are offered to these ROMs.

17.4.1 Paged ROM service calls

The mechanism by which a service call is performed is as follows. The operating system pages in each paged ROM in turn starting with the ROM in the highest priority socket (paging is performed by writing to a value to a hardware latch, the hardware responds to the value written to this location and performs the relevant switching of the chip select signals). If the ROM has a service entry point then this code is executed. Before entering the code the accumulator is loaded with a reason code. On entry, the X register will contain the current ROM number (a ROM is thus able to tell which socket it is in) and the Y register will be loaded with any further relevant information. The paged ROM can return control to the operating system following an RTS instruction. If the ROM has responded and does not wish any further action to be taken then the accumulator should be set to zero, otherwise all registers should be unchanged.

Below is a list of the reason codes which may be presented to a paged ROM when a service call is performed.

Reason code &00 No Operation

No operation, this service call should be ignored because a higher priority ROM has already claimed it.

Reason code &01 Absolute Public Workspace Claim

This call is made during a reset. The operating system is interrogating each ROM to determine how much workspace memory would be required if that ROM were called. This workspace is available temporarily while the ROM is active. Pages &E00 and above are available as a fixed area on the BBC micro and the Electron. Each paged ROM is entered with A=&01, X=ROM number and Y=top of fixed area. For the highest priority ROM on a BBC micro the Y register will contain &0E. The Y register value should be increased in accordance with the requirements of the ROM. If the Y register setting is sufficient or greater than that required then the service routine should return the Y register unaltered.

Reason code &02 Relative Space Claim

This call is made by the operating system during a reset to determine how much private RAM workspace is required by each ROM. The position of this private area will start from the top of the absolute space claimed by the ROMs and on the relative space claimed by higher priority ROMs. This call is made with the Y register containing the value of the first available page. This value should be stored in the ROM workspace table at &DF0 to &DFF (for ROMs 0 to 15 respectively) and the Y register returned with its value increased by the number of pages of private workspace required.

Reason code &03 Auto-boot call

This call is issued during a reset to allow each service ROM to initialise itself. It enables the highest priority filing system to set up its vectors automatically rather than require explicit selection with a *command. To allow lower priority services to be selected, the service ROM should examine the keyboard and initialise only if either no key is pressed or if its own ROM specific key is being pressed (e.g. D+BREAK for Acorn DFS). If the ROM initialises it should attempt to look for a boot file (typically !BOOT) to RUN, EXEC or LOAD if the Y register contains zero. This call is made during a reset after the start-up messages have been printed.

Reason code &04 Unrecognised *command

When a line of text is offered to the command line interpreter (CLI), the operating system will pass on any unrecognised command to each of the paged ROMs and then, if still unrecognised to the currently active filing system. When an unrecognised command is offered to the paged ROMs this service call is made.

Entry parameters:

A=&04
X=ROM number
Y contains an offset which if added to the contents of &F2 and &F3 point to the beginning of the text with the asterisk and leading spaces stripped off and terminated with a carriage return

On exit:

Registers restored
A=0 if recognised

Filing systems should not intercept filing system commands (which will be common to all filing systems) using this service call but may intercept some filing system utilities (e.g. *DISC, *NET).

Reason code &05 Unknown interrupt

An interrupt which is not recognised by the operating system or which has been masked out by software will result in this call being generated. A service ROM that services devices which might cause interrupts should interrogate such devices to determine if they have generated the unrecognised interrupt. If the interrupt is then recognised and processed, the accumulator should be returned with zero to prevent other ROMs being offered the interrupt. The routine should terminate with an RTS not an RTI.

Reason code &06 BRK has been executed

If a BRK instruction is encountered this call will be generated before indirection occurs through the BRK vector (BRKVEC, &202). BRKs are usually used to indicate that an error condition has occurred.

Entry parameters:

A=&06
X=ROM number
Y is undefined but should be preserved
&F0 contains the value of the stack pointer

&FD and &FE point to the error number which is not necessarily in the current ROM (OSBYTE &BA yields this ROM number)

On exit:

All registers should be preserved

Reason code &07 Unrecognised OSBYTE call

When an OSBYTE call has been made and is not recognised by the operating system, it is offered to the paged ROMs by this service call. The contents of the A, X and Y registers at the time of the OSBYTE call are stored in locations &EF, &F0 and &F1 respectively.

Reason code &08 Unrecognised OSWORD call

This service call is performed in response to the user issuing an OSWORD call not catered for in the operating system. The contents of the A, X and Y registers at the time of the call are stored in locations &EF, &F0 and &F1 respectively. Unrecognised OSWORD calls with accumulator values greater than or equal to &E0 are offered to the user vector (USERV, &200). An OWORD call with A=7 (equivalent to the SOUND command in BASIC) which has been given an unrecognised channel will also generate this service call.

Reason code &09 *HELP command interception

This service call is generated when the *HELP command is passed through the CLI. The remainder of the command line is pointed to by the address stored in locations &F2 and &F3 plus an offset in Y. Each ROM is required to respond to this call. If the remainder of the command line is blank ,the ROM should print its name and version number followed by a list of subheadings to which the ROM will respond.

e.g. Acorn DFS (version 0.90) outputs:

```
DFS 0.90
  DFS
  UTILS
```

Indicating that this ROM responds to *HELP DFS and *HELP UTILS

If the rest of the command line is not blank, the service routine should compare it against its subheadings and if a match occurs should output the information under that subheading.

e.g. Acorn DFS responds to *HELP UTILS with:

```
DFS 0.90
   BUILD <fsp>
```

```
DISC
DUMP <fsp>
TYPE <fsp>
```

If there is more than one item on a line then the ROM should deal with these items individually. All registers should be preserved across the service routine.

Reason code &0A Claim absolute workspace

This service call originates from a paged ROM which requires the use of the absolute workspace. When a ROM is active and requires use of this workspace it should issue an OSBYTE call &8F with X=&0A which will generate this service call. The previous owner of the absolute workspace is then able to save any valuable contents of this memory in its own private data area in the relative workspace. The previous owner should also update a flag within its private data area indicating that it no longer owns the absolute workspace. The active filing system is selected independently of the ownership of the absolute workspace. Thus while a filing system ROM may have ownership of this workspace the tape filing system may be selected (the tape FS does not require any absolute workspace). Problems may arise when the active filing system paged ROM is called upon but does not have ownership of the absolute workspace. The active filing system should then issue this service call to obtain the use of the absolute workspace.

Reason code &0B NMI released

This service call also originates from paged ROMs and should be generated by using OSBYTE call &8F. This call should be issued when a ROM no longer requires the NMI. This releases the zero page locations &A0 to &A7 and the space for the NMI routine in page &D00. On entry the Y register contains the ROM number of the previous owner and this should be compared to the ROM's own identity before reasserting control of the NMI.

Reason code &0C NMI claim

This call should be generated by a paged ROM using OSBYTE &8F when it wishes to take possession of the NMI. The service call should be generated passing &FF in the Y register (i.e. OSBYTE A=&8F, X=&0C and Y=&FF). The current owner should relinquish control by returning its ROM number in the Y register in response to this call.

Reason code &0D ROM filing system initialise

This call is issued when the ROM filing system (RFS) has been activated in response to a *ROM command and is now searching for a file. On entry the Y register contains 15 minus the ROM number of the next ROM to be scanned. If this ROM number is less than the current ROM's ID this call should be ignored. Otherwise the active ROM number should be stored in &F5 (in the form 15-ROM number) where the RFS active ROM number is stored. The current ROM should indicate that the service call has been claimed by returning zero in the accumulator and should store a pointer to the data stored within the ROM in locations &F6 and &F7 set aside for use by the RFS.

See chapter 17.5.7 for an example of an RFS ROM.

Reason code &0E ROM filing system get byte

This service call may be issued after a ROM containing RFS data has been initialised with service call &0D. A ROM should respond only if it is the active RFS ROM as indicated by the value in location &F5 (stored in the form 15-ROM number). The fetched byte should be returned in the Y register.

See chapter 17.5.7 for an example of an RFS ROM.

Reason code &0F Vectors claimed

This service call should be generated by any paged ROM (using OSBYTE &8F) which has been initialised and subsequently changes any operating system vector. This call warns paged ROMs that a change has occurred.

Reason code &10 Close any *SPOOL or*EXEC files

This call is also issued by filing systems during their initialisation to allow the previously selected filing system to close any open spool or exec files.

Reason code &11 Font implosion/explosion warning

In the Electron or BBC model B, when OSBYTE &14 is used to change the RAM allocation for user defined characters, this service call is issued. This call is issued to warn languages that the OSHWM has been changed and thus the user RAM allocation has also changed. The font is permanently exploded in the Master series computers.

Reason code &12 Initialise filing system

This call enables third party software to switch between one or more filing systems without having to issue *commands. A program may want to switch between two filing systems in order to transfer files. A filing system ROM should respond to this call if the value in the Y register corresponds to its ROM number. All filing systems should allow files to be open while inactive and on receiving this call should restore any such files.

Reason code &13 Character placed in RS423 buffer

This call is made when the Electron OS has placed a character in the RS423 buffer. Expansion software handling RS423 hardware should respond to this call. If not claimed the operating system purges the RS423 buffer.

This service call is not implemented on other Acorn BBC series machines.

Reason code &14 Character placed in printer buffer

This call is made when the Electron OS has placed a character in the printer buffer. Expansion software controlling printer hardware should respond to this call.

This service call is not implemented on other Acorn BBC series microcomputers.

Reason code &15 100 Hz poll

The Master series or Electron operating systems will provide a 100 Hz polling call for the use of paged ROMs. A paged ROM requiring this call should increment the polling semaphore using OSBYTE &16 (22) on initialisation and decrement it using OSBYTE &17 (23) when polling is no longer required. The operating system will issue this service call when the semaphore is non-zero. The semaphore itself may be read using OSBYTE &B9 (185). This facility is implemented mainly so that hardware devices may be supported as a background task without being interrupt driven. This would be suitable for hardware which does not require urgent servicing. The Y register contains the semaphore value which should be decremented by the service routine by the number of times that the service ROM nmakes use of it.

This service call is not implemented on the BBC model B.

Reason code &16 A BEL request has been made

When the external sound flag (OSBYTE &DB/219) is set this call is issued by the Electron OS in response to an ASCII BEL code being written (VDU 7). This is to enable the external sound system to respond appropriately.

This service call is only implemented on the Electron.

Reason code &17 SOUND buffer purged

This call is made when an external sound system is flagged on the Electron and an attempt has been made to purge any of the SOUND buffers.

This service call is only implemented on the Electron.

Reason code &18 Interactive *HELP request

This call is implemented on the Master series computers only where the ANFS responds to this call by typing a text file from disc. Otherwise this call is issued following a call with A=9 and appears to operate in the same manner.

Reason code &21 Claim private workspace in Hazel

This call is available only on the Master series computers. This call operates in the same way as service call &01 but memory is allocated in the filing system shadow RAM from &C000-&DCFF. No claims should be made for workspace outside this range. If further workspace is required a response should be made following the service call &01 which follows this call.

Reason code &22 Claim absolute workspace in Hazel

This call is available only on the Master series computers and operates in the same way as service call &02 and is subject to the same additional restraints as service call &21.

Reason code &23 Reports top of private workspace

This call is available only on the Master series computers. This call reports the limit of private workspace claimed in the filing system shadow RAM to allow filing systems to make use of the surplus memory as temporary workspace if available.

Reason code &24 Absolute workspace in Hazel count

This call is only issued by the Master series computers. The value of Y issued by the call should be decremented by the number of pages of absolute workspace a filing system would like to claim.

Reason code &25 Return filing system information

This call is issued by the Master series computers in order to provide the filing system handler with information to allow the use of temporary filing systems. The 11 byte filing system information block as described in section 16.2 should be written to at the address pointed to by (&F2),Y.

Reason code &26 *SHUT command issued

The Master series operating system issues this call following a *SHUT command and filing systems should respond by selecting itself and closing all files. The current filing system will be reselected by the operating system.

Reason code &27 Hard Reset call

This call is issued following a power on reset or a control break and is only available on the Master series computers.

Reason code &28 Unknown *CONFIG. command

Available on on the Master or Master Compact this call enables ROMs to add options to the *CONFIGURE command. The remaining command line is pointed to by (&F2),Y.

Reason code &29 Unknown *STATUS command

As for service &28 this call enables ROMs to respond to the corresponding *STATUS commands.

Reason code &2A Language about to be initialised

This call is issued by the Master series computers and is intended to allow the activation of any language support ROMs.

Reason code &2C Compact Joystick call

This call is used on the Master Compact to allow paged ROMs to respond to devices attatched to the user port.

On entry:

Y contains offset from &200 pointing at a parameter block

&200+Y+0	ADVAL least significant byte
&200+Y+1 &200+Y+2	X co-ordinate, LSB first
&200+Y+3 &200+Y+4	Y co-ordinate, LSB first
&200+Y+5 &200+Y+6	2 spare bytes

Reason code &FE Post initialisation Tube system call

The operating system makes this call during a reset after the OSHWM has been set. The Tube service ROM responds to this by exploding the user defined character RAM allocation.

Reason code &FF Tube system main initialisation

This call is issued only if the Tube hardware has been detected. This call is made prior to message generation and filing system initialisation.

The fact that these calls are shared by all the service ROMs can lead to widespread consequences if a service call is misused by one of the ROMs. The programmer should consider the consequences of his ROM claiming calls (or not claiming calls) when present.

17.4.2 Service ROM example

The program below is a ROM based version of a printer buffer program. This program may be assembled and then loaded into sideways RAM for testing or blown onto an EPROM. The example is offered as a complete example of a service type ROM to pull together and illustrate some of the descriptions given above.

```
10 REM Assembler program printer buffer ROM
 20 DIM code% &400
 30 INSV=&22A:nI=&2A/2
 40 RMV=&22C:nR=&2C/2
 50 CNPV=&22E:nC=&2E/2
 60 ptrblk=&70
 70 ip_ptr=ptrblk+2
 80 op_ptr=ptrblk+4
 90 old_bfr=&880
100 begin=old_bfr
110 end=old_bfr+2
120 wrkbt=old_bfr+4
130 size=old_bfr+5
140 vec_cpy=old_bfr+6
150 line=&F2
160 OSASCI=&FFE3
170 OSBYTE=&FFF4

180 FOR I=4 TO 7 STEP 3
190 P%=&8000:O%=code%
200 [
210 OPT I

220 .romstrt EQUB 0                    \ null language entry point
230          EQUB 0
240          EQUB 0
250          JMP service               \ service entry point
260          EQUB &82                  \ ROM type byte, service ROM
270          EQUB (copyr-romstrt)      \ offset to copyright string
280          EQUB 0                    \ null byte
290 .title   EQUB &A                   \ title string
300          EQUS "BUFFER"
310          EQUB &0                   \ null byte
320          EQUS "1.00"               \ version string
330          EQUB &D                   \ carriage return
340 .copyr   EQUB 0                    \ terminator byte
350          EQUS "(C)1984 Mark Holmes" \ copyright message
360          EQUB 0                    \ terminator byte

370 \ End of ROM header, start of code

380 .name    EQUS "REFFUB"             \ command name

390 \ Service handling code, A=reason code, X=ROM id & Y=data

400 .service CMP #4                    \ is reason unknown command?
410          BEQ command               \ if so goto 'command'
```

```
420           CMP #9                  \ is reason *HELP
430           BEQ help                \ if so goto 'help'
440           CMP #2                  \ is reason private wrkspace
450           BEQ wkspclm             \ if so goto 'wkspclm'
460           CMP #3                  \ is reason autoboot call
470           BNE notboot             \ if NOT goto 'notboot'
480           JMP autorun             \ BEQ autorun, out of range
490 .notboot  RTS                     \ other reason, pass on

500 \ Unknown command, is it *BUFFER ?
510 \ (command line address in &F2,&F3 (line) + offset Y)

520 .command  TYA:PHA:TXA:PHA         \ save registers
530           LDX #6                  \ X=length of name
540 .loop1    LDA (line),Y            \ A=next letter of command
550           CMP name-1,X            \ compare with my name
560           BNE notme               \ not equal, goto 'notme'
570           INY                     \ for next letter of command
580           DEX                     \ for next letter of name
590           BNE loop1               \ if X<>0 round again
600           BEQ parmch              \ 6 letters matched, do jump
610 .notme    PLA:TAX:PLA:TAY         \ no match, restore registrs
620           LDA #4                  \ restore reason code
630           RTS                     \ pass on call

640 \ *HELP response (parameters as for call above)

650 .help     TYA:PHA:TXA:PHA         \ save registers
660           LDX #0                  \ use X as index counter
670 .loop2    LDA title,X             \ A=next letter from title $
680           BNE over1               \ if A<>0 jump next instrctn
690           LDA #&20                \ replace 0 by space char.
700 .over1    JSR OSASCI              \ write character
710           INX                     \ increment index counter
720           CPX #(copyr-title)      \ end of title ?
730           BNE loop2               \ if not get another char.
740           PLA:TAX:PLA:TAY         \ restore registers
750           LDA #9                  \ restore A
760           RTS                     \ pass on *HELP call

770 \ Oportunity to claim private workspace
780 \ (Y=1st page no. free, call inc's Y by no. pages claimed)

790 .wkspclm  TYA                     \ copy page no. to A
800           STA &DF0,X              \ table for ROMs' workspace
810           PHA                     \ save page no. on stack
820           LDA #&FD
830           LDX #0
840           LDY #&FF                \ OSBYTE call to read last
850           JSR OSBYTE              \  BREAK type
860           CPX #0                  \ X=0 after soft reset
870           BEQ softrst             \ soft brk, dont reset size
880           LDA #8                  \ 8 pages for printer buffr
890           STA size                \ location for buffer size
900 .softrst  CLC                     \ clear carry, for add
910           PLA                     \ original Y on stack
920           ADC size                \ A=A+?size
930           TAY                     \ Y=A
940           LDX &F4                 \ X=ROMid
950           LDA #2                  \ restore A (reason code)
960           RTS                     \ pass on workspace call
```

```
 970 \ *BUFFER command issued, reset buffer size

 980 .parmch  LDA (line),Y          \ get char. from cmnd line
 990          CMP #&D               \ car.ret.? end of line ?
1000          BNE ok_init           \ if not, cont. line input
1010          LDA #1                \ no parameters so set
1020          JMP default           \  default buffer size
1030 .ok_init INY                   \ increment index counter
1040          CMP #&20              \ was char. a space?
1050          BEQ parmch            \ if so get next character
1060          SEC                   \ set carry for subrtact
1070          SBC #&30              \ A=A-ASC"0"
1080          CMP #0                \ was character zero
1090          BEQ deinit            \ if so, switch off
1100          BMI rngerr            \ char.<0, out of range
1110          CMP #6                \ compare char. to 6
1120          BPL rngerr            \ A>=6, out of range
1130 .default CLC                   \ clear carry for ASL
1140          ASL A:ASL A:ASL A     \ A=A*8
1150          STA size              \ store for buffer size
1160 .prntmes LDA #&87              \ Use OSBYTE &87 to read
1170          JSR OSBYTE            \  current screen MODE
1180          TYA                   \ A=Y
1190          TAX                   \ X=A
1200          LDY #&F8              \ Use OSBYTE &FF to write
1210          LDA #&FF              \  MODE selected on reset
1220          JSR OSBYTE            \  (i.e. MODE preserved)
1230          TAX                   \ X=&FF
1240 .loop6   INX                   \ increment index counter
1250          LDA message,X         \ A=next byte of message
1260          JSR OSASCI            \ print character
1270          CMP #&D               \ was it carriage return
1280          BNE loop6             \ if not get next character
1290          PLA:TAX:PLA:TAY       \ restore registers
1300          LDA #0                \ claim call, 0 reason code
1310          RTS                   \ return
1320 .message EQUB &A               \ message string
1330          EQUS "Press BREAK to change buffer size"
1340          EQUB &D
1350 .rngerr  LDX #&FF              \ set index counter
1360 .loop7   INX                   \ increment index counter
1370          LDA errdata,X         \ A=character from string
1380          STA &100,X            \ copy to bottom of stack
1390          CMP #&FF              \ was byte terminator
1400          BNE loop7             \ if not loop again
1410          JMP &100              \ goto &100 (BRK)
1420 .errdata EQUB 0                \ BRK opcode
1430          EQUB 0                \ error number 0
1440          EQUS "Invalid buffer size" \error message
1450          EQUB 0                \ message string end
1460          EQUB &FF              \ terminator byte

1470 \ Routine for deselecting buffer ROM routines

1480 .deinit  LDA #3                \ VDU3, just in case
1490          JSR OSASCI
1500          SEI                   \ disable interrupts
1510          LDY #0
1520          STY size              \ size=0
1530 .loop8   LDA vec_cpy,Y         \ load old vector contents
1540          STA INSV,Y            \ store in vector
```

```
1550            INY                     \ increment index counter
1560            CPY #6                  \ copied 6 bytes yet
1570            BNE loop8               \ if not loop again
1580            CLI                     \ enable interrupts
1590            JMP prntmes             \ print message + return

1600 \ Initialise buffer routines automatically

1610 .autorun TYA:PHA:TXA:PHA           \ preserve registers
1620            LDA size                \ A=buffer size in pages
1630            BEQ no_init             \ A=0, don't initialise
1640            LDA #&84                \ HIMEM OSBYTE number
1650            JSR OSBYTE              \ make call
1660            STY end                 \ store page address
1670            LDA #&83                \ OSHWM OSBYTE number
1680            JSR OSBYTE              \ make call
1690            CPY end                 \ is OSHWM > HIMEM
1700            BCC room                \ if so continue
1710            JMP no_room             \ no room so cause error
1720 .room      JSR init                \ call initialise routine
1730 .no_init   PLA:TAX:PLA:TAY         \ restore registers
1740            LDA #3                  \ restore A
1750            RTS                     \ return
1760 .init      LDA #&A8
1770            LDX #0
1780            LDY #&FF                \ OSBYTE to read address of
1790            JSR OSBYTE              \  extended vector table
1800            STX ptrblk              \ set up zero page locations
1810            STY ptrblk+1            \  for indirect indexed adr.
1820            LDY #3*nI               \ offset into table (INSV)
1830            LDA #ins AND &FF        \ address of new routine
1840            SEI                     \ disable interrupts
1850            STA (ptrblk),Y          \ copy address to vector
1860            INY                     \ Y=Y+1
1870            LDA #ins DIV &100       \ high byte of address
1880            STA (ptrblk),Y          \ copy to extended vector
1890            INY                     \ Y=Y+1
1900            LDA &F4                 \ A=ROMid
1910            STA (ptrblk),Y          \ complete extended vector
1920            INY                     \ Y=Y+1
1930            LDA #rem AND &FF        \ REMV new routine address
1940            STA (ptrblk),Y          \ lo byte to extended vector
1950            INY                     \ Y=Y+1
1960            LDA #rem DIV &100       \ Hi byte of new routine
1970            STA (ptrblk),Y          \ place in extended vector
1980            INY                     \ Y=Y+1
1990            LDA &F4                 \ A=ROMid
2000            STA (ptrblk),Y          \ complete REMV 3 byte vect.
2010            INY                     \ Y=Y+1
2020            LDA #cnp AND &FF        \ repeat, store address of
2030            STA (ptrblk),Y          \  new CNPV routine in the
2040            INY                     \  extended vector together
2050            LDA #cnp DIV &100       \  with ROM number.
2060            STA (ptrblk),Y
2070            INY
2080            LDA &F4
2090            STA (ptrblk),Y
2100            TAX                     \ X=ROMid
2110            LDY #0                  \ Y=0
2120 .loop3     LDA INSV,Y              \ A=old vector contents
2130            STA vec_cpy,Y           \ copy to workspace
```

```
2140            INY                     \ Y=Y+1
2150            CPY #6                  \ copied 6 bytes yet ?
2160            BNE loop3               \ if not loop again
2170            LDA &DF0,X              \ A=workspace addr. hi byte
2180            STA begin+1             \ store in zero page
2190            CLC                     \ clear carry for add
2200            ADC size                \ add begin+size
2210            STA end+1:DEC end+1     \ store in zero page, -1
2220            LDY #&10                \ lo byte of begin
2230            STY begin               \  (room for return vect's)
2240            LDY #&FF                \ lo byte of end
2250            STY end                 \ store in zero page
2260            JSR rstptrs             \ reset ip+op ptrs
2270            LDA #nI*3               \ for the extended vector
2280            STA INSV                \  system the vectors must
2290            LDA #nR*3               \  now point to &FF00 +
2300            STA RMV                 \  vector number*3
2310            LDA #nC*3
2320            STA CNPV
2330            LDA #&FF
2340            STA INSV+1
2350            STA RMV+1
2360            STA CNPV+1
2370            CLI                     \ enable interrupts
2380            RTS                     \ return
2390 .no_room   CLI                     \ clear interrupts
2400            LDA #&7F                \ disable system VIA this
2410            STA &FE4E               \   makes BREAK power on rst
2420            LDX #&FF                \ set up index counter
2430 .loop9     LDA nrmerr,X            \ fetch next byte of data
2440            STA &100,X              \ store at bottom of stack
2450            INX                     \ increment index counter
2460            CMP #&FF                \ reached terminator ?
2470            BNE loop9               \ if not loop again
2480            JMP &100                \ execute BRK (not in ROM)
2490 .nrmerr    EQUB 0                  \ BRK instruction opcode
2500            EQUB 0                  \ error number 0
2510        EQUS "Not enough room for print buffer, Press BREAK"
2520            EQUB 0                  \ string terminator
2530            EQUB &FF                \ data end

2540 \ Purge buffer by setting i/p + o/p ptrs to buffer start

2550 .rstptrs LDA begin                 \ lo byte bufr start address
2560            STA ip_ptr              \ store input pointer
2570            STA op_ptr              \ store output pointer
2580            LDA begin+1             \ hi byte of buffer start
2590            STA ip_ptr+1            \ store input pointer
2600            STA op_ptr+1            \ store output pointer
2610            RTS                     \ return

2620 .wrngbf1 PLA:PLP:JMP (vec_cpy) \ old INSV routine

2630 \ New insert char. into buffer routine

2640 .ins       PHP:PHA                 \ save S and A on stack
2650            CPX #3                  \ is buffer id 3 ?
2660            BNE wrngbf1             \ if not pass to old routine
2670            PLA:PLP:PHA             \ not passing on, tidy stack
2680            LDA ip_ptr              \ A=lo byte of input pointer
2690            PHA                     \ store on stack
```

```
2700            LDA ip_ptr+1            \ A=hi byte of input pointer
2710            PHA                     \ store on stack
2720            LDY #0                  \ Y=0 so ip_ptr incremented
2730            JSR inc_ptr             \  by the inc_ptr routine
2740            JSR compare             \ compare the two pointers
2750            BEQ insfail             \ if ptrs equal, buffer full
2760            PLA:PLA:PLA             \ don't need ip_ptr copy now
2770            STA (ip_ptr),Y          \ A off stack, insrt in bufr
2780            CLC                     \ insertion success, C=0
2790            RTS                     \ finished
2800 .insfail   PLA                     \ buffer was full so must
2810            STA ip_ptr+1            \  restore ip_ptr which was
2820            PLA                     \  stored on the stack
2830            STA ip_ptr
2840            PLA
2850            SEC                     \ insertion fails so C=1
2860            RTS                     \ finished

2870 .wrngbf2 PLP:JMP (vec_cpy+2)   \ old REMV routine

2880 \ New remove char. from buffer routine

2890 .rem       PHP                     \ save status register
2900            CPX #3                  \ is buffer id 3 ?
2910            BNE wrngbf2             \ if not use OS routine
2920            PLP                     \ restore status register
2930            BVS examine             \ V=1, examine not remove
2940 .remsr     JSR compare             \ compare i/p and o/p ptrs
2950            BEQ empty               \ if the same, buffer empty
2960            LDY #2                  \ Y=2 so that increment ptr
2970            JSR inc_ptr             \  routine inc's op_ptr
2980            LDY #0                  \ Y=0, for next instruction
2990            LDA (op_ptr),Y          \ fetch character from bufr
3000            TAY                     \ return it in Y
3010            CLC                     \ buffer not empty, C=0
3020            RTS                     \ return
3030 .empty     SEC                     \ buffer empty, C=1
3040            RTS                     \ return
3050 .examine   LDA op_ptr              \ examine only, so store a
3060            PHA                     \  copy of the o/p pointer
3070            LDA op_ptr+1            \  on the stack to restore
3080            PHA                     \  ptr after fetch
3090            JSR remsr               \ fetch byte from buffer
3100            PLA                     \ restore ptr from stack
3110            STA op_ptr+1            \ (if buffer was empty
3120            PLA                     \  C=1 from fetch call)
3130            STA op_ptr
3140            TYA                     \ examine requires ch. in A
3150            RTS                     \ finished

3160 .wrngbf3 PLP:JMP (vec_cpy+4)   \ old CNPV routine

3170 \ New count/purge buffer routine

3180 .cnp       PHP                     \ save status reg. on stack
3190            CPX #3                  \ is buffer id 3 ?
3200            BNE wrngbf3             \ if not pass to old subr
3210            PLP                     \ restore status register
3220            PHP                     \ save again
3230            BVS purge               \ if V=1, purge required
3240            BCC len                 \ if C=0, amount in buffer
```

```
3250            LDA ip_ptr              \  o/w free space request
3260            PHA
3270            LDA ip_ptr+1            \ store ip_ptr on stack
3280            PHA
3290            LDX #0                  \ X=0 for use as counter
3300            STX wrkbt               \ wrkbt=0 for hi counter
3310            LDY #0                  \ Y=0, so ip_ptr incr'd
3320 .loop1     JSR inc_ptr             \ increment ip_ptr
3330            JSR compare             \ does it equal op_ptr
3340            BEQ finshd1             \ if so count=free space
3350            INX                     \ X=X+1
3360            BNE no_inc              \ if X=0 don't inc wrkbt
3370            INC wrkbt               \ hi byte of count inc'd
3380 .no_inc    JMP loop1               \ loop round again
3390 .finshd1   PLA                     \ restore ip_ptr off stack
3400            STA ip_ptr+1
3410            PLA
3420            STA ip_ptr
3430            LDY wrkbt               \ Y=hi byte of free space
3440            PLP                     \ restore status register
3450            RTS                     \ finished
3460 .len       LDA op_ptr              \ store op_ptr on stack
3470            PHA
3480            LDA op_ptr+1
3490            PHA
3500            LDX #0                  \ X=0 for use as counter
3510            STX wrkbt               \ wrkbt=0 hi byte of count
3520            LDY #2                  \ Y=2 so op_ptr incremented
3530 .loop2     JSR compare             \ are ptrs equal ?
3540            BEQ finshd2             \ if so buffer empty
3550            JSR inc_ptr             \ increment op_ptr
3560            INX                     \ increment count
3570            BNE no_inc2             \ if X=0 then increment hi
3580            INC wrkbt               \  byte of count
3590 .no_inc2   JMP loop2               \ loop round again
3600 .finshd2   PLA                     \ restore op_ptr off stack
3610            STA op_ptr+1
3620            PLA
3630            STA op_ptr
3640            LDY wrkbt               \ Y=hi byte of length
3650            PLP                     \ restore status register
3660            RTS                     \ finished
3670 .purge     JSR rstptrs             \ reset i/p & o/p pointers
3680            PLP                     \ restore status register
3690            RTS                     \ return

3700 \ Increment pointer routine. Y=0 op_ptr, Y=2 ip_ptr

3710 .inc_ptr CLC                       \ clear carry for add
3720            LDA ip_ptr,Y
3730            ADC #1
3740            STA ip_ptr,Y
3750            LDA ip_ptr+1,Y
3760            ADC #0
3770            STA ip_ptr+1,Y          \ pointer=pointer+1
3780            CMP end+1               \ hi byte reached buffr end?
3790            BNE home                \ if not finish
3800            LDA ip_ptr,Y
3810            CMP end                 \ lo byte reached end ?
3820            BNE home                \ if not finish
3830            LDA begin               \ reached end of buffer
```

```
3840           STA ip_ptr,Y            \  so reset pointer to
3850           LDA begin+1             \  start address of buffer
3860           STA ip_ptr+1,Y
3870.home      RTS                     \ return

3880 \ Compare pointers. if equal Z=1 don't care otherwise

3890 .compare  LDA ip_ptr+1
3900           CMP op_ptr+1            \ compare ptr high bytes
3910           BNE return              \ if not equal return
3920           LDA ip_ptr
3930           CMP op_ptr              \ compare pointer low bytes
3940 .return   RTS                     \ return
3950 ]
 3960 NEXT

3970OSCLI"*S.BRM "+STR$~code%+" "+STR$~O%
```

This program may be run to assemble a ROM image which can be loaded into sideways RAM or blown into an EPROM. When this ROM is present in a machine an enlarged printer buffer of 2k is automatically initialised following a reset. Typing '*BUFFERn' with n from 1 to 5 selects a buffer size of n*2K ,next time a reset occurs. '*BUFFER0' deselects the enlarged buffer and re-initialises the normal OS routines. '*BUFFER' (no parameters) reselects the default buffer size (2K).

17.4.3 Extended Vectors

In the example above the operating system buffer maintenance vectors had to be set to point to routines held within the service ROM. The operating system supports a system of extended vectors to enable each of the OS vectors to point to routines held in paged ROMs.

Each OS vector is identified by a number which may be calculated by subtracting &200 (the vector space base address) from the vector address and dividing by two (each vector is two bytes).

The extended vectors themselves are three byte values stored in the extended vector space (the address of which is returned by OSBYTE &A8). Each extended vector consists of a 2 byte address followed by a byte containing the ROM number.

An extended vector routine is reserved for the use of each of the operating system vectors and uses an assigned extended vector. These routines are located in the operating system starting at &FF00. To calculate the calling address of the reserved extended vector call the vector number multiplied by three should be added to &FF00. The extended vector routines use The address of the extended vector used

by this routine is given by multiplying the vector number by three and adding the vector space address.

The procedure for a paged ROM to intercept a vector is:

(a) Determine buffer number n
(b) Establish extended vector space, V using OSBYTE &A8
(c) Store new routine's address in (V+3*n)
(d) Store ROM number following address
(e) Make copy of OS vectors contents if required for return
(f) Store address (&FF00+3*n) in OS vector (&200+2*n)

It is usually a good idea to disable interrupts during this change-over so that an interrupt routine is not able to use the vector during the change.

17.5 Serially accessed ROMs & the *ROM filing system

The BBC microcomputer and the Electron have been designed to use software contained in ROM cartridge packs. The ROM packs which plug into paged ROM sockets may contain up to about 16K of data and/or programs. On the BBC microcomputer the facility also extends to phrase ROMs (PHROMS) associated with the speech upgrade. When the programs or data stored in these ROM packs are required it may be loaded into user RAM in the same way as programs or data may be loaded off tape or disc.

These ROM packs are intended to provide a reliable and rapidly accessible medium for the distribution of programs. This product is very useful for owners of tape based machines who would otherwise have to rely upon the much slower and inherently less reliable medium.

The advantage to the software producer is that there is no requirement for a special version of the program to be written. A system is required for the formatting of the program for inclusion in a ROM pack but there is no need to modifiy the program itself.

The *ROM filing system is a subset of the tape filing system. Paged ROMs are interrogated to determine whether they contain information intended for this filing system and are then serially accessed by the *ROM filing system.

Paged ROMs containing information intended for access via the *ROM filing system are no different from other paged ROMs. They are service type ROMs and as such have service entry points. They are distinguishable as *ROM filing system ROMs only by their response to paged ROM service calls issued by the *ROM filing system code. When the user selects the *ROM filing system any further requests for files result in the *ROM filing system section of the operating system scanning the paged ROMs for these files. A paged ROM containing files intended for the *ROM filing system should respond to one of two paged ROM service calls.

The two service calls and the responses expected from ROMs containing *ROM data are described in detail below. One call expects the ROM to prepare to yield any data it has and the second call is used to extract this data, one byte at a time. The data should be formatted in a similar way to the data stored on tape but is modified in such a way as to minimise the storage overheads involved in using such a format. The reason for adopting this format is to minimise the requirements for extra code in the operating system while utilising the exhaustive error checking already in existence. Accompanying these advantages there is a concurrent reduction in response time performance but this is of little importance to the users of tape based machines who are still able to appreciate a substantial improvement on their system's existing performance.

17.5.1 Converting files to *ROM format

In order to produce a ROM containing files which will be recognised by the *ROM filing system it is necessary to fulfil two criteria. The first requirement is for some header code which will recognise the *ROM filing system paged ROM service calls and respond accordingly. The second requirement is that the data which makes up the files is formatted in the manner in which the *ROM filing system expects to find it.

17.5.2 The header code

As has been stated above a paged ROM which is to be recognised by the *ROM filing system is a perfectly standard paged ROM which responds to the appropriate service calls. As a result of this requirement the first part of each *ROM filing system ROM consists of a standard format paged ROM header followed by a small amount of code which responds to the necessary service calls. By convention *ROM paged ROMs do

not respond to the *HELP service call since the space occupied by the necessary code would leave less space for programs and data.

The two paged ROM service calls which should elicit a response from *ROM paged ROMs are described in the next two sections.

17.5.3 Paged ROM service call with A=&D

This call is the *ROM filing system initialise call. When the filing system is active and wishes to scan the next ROM this call is issued.

The initialise service call is made with the ROM number of the next ROM to be scanned in the Y register. Having received this service call a filing system ROM should only respond if its own ROM id (stored in location &F4) is greater than or equal to the ROM number passed in the Y register.

Having decided to claim this service call the ROM should place its own ROM number in location &F5 which marks it as the currently active *ROM filing system ROM. It should then write the address of the start of its data in locations &F6 and &F7. This provides a zero page pointer which is used by the filing system code to extract bytes of data serially from the ROM.

Having performed these two operations the service routine should return with the accumulator containing zero to indicate that the call has been claimed. In the case of the paged ROM id being less than the ROM number in the Y register the service routine should exit with &D in the accumulator and the operating system will then offer the call to the next ROM.

The actual mode in which the *ROM filing system ROM numbers are represented differs from the way in which the paged ROM id's are usually represented (i.e. as stored in &F4, a number 0 to 15). The filing system ROM numbers are represented by a value which is 15 minus the physical paged ROM number. One way of converting numbers from one form to another is given below.

```
EOR #&FF
AND #&F
```

The number to converted is passed in the accumulator. This code performs the necessary conversion and returns the result in the accumulator. These instructions will always convert a number into the other representation.

17.5.4 Paged ROM service call with A=&E

Having obtained a response from a paged ROM to service call &D the *ROM filing system will use this service call to read bytes from the data contained in the ROM.

There is a difference in how the service routine can be implemented on the BBC Microcomputer OS 1.00 and later OS versions (including the Electron). The actual response required from the service call is essentially the same however.

When called by OS 1.00 a paged ROM should only respond to this call if its own ROM id is the same as the current *ROM filing system ROM number. A comparison of the contents of memory location &F4 (current paged ROM) should be made with the inverted contents of &F5 (current *ROM). If these are not the same the call should be returned unclaimed.

The service routine for OS 1.00 should return the byte of data pointed to by the pointer in &F6 and &F7, in the Y register (e.g. `LDA (&F6),Y:TAY`) and increment this pointer so that it is ready for the next call.

Later operating system versions contain a routine (OSRDRM) which given the paged ROM id of the current *ROM filing system ROM in the Y register will read a byte from this paged ROM using the pointer at &F6+&F7. Thus this paged ROM service call may be serviced by the highest priority *ROM filing system ROM and the operating system does not have to scan all the ROMs before getting a response. This leads to a significant improvement in performance of the *ROM filing system.

The service routines are able to determine which operating system has called them by the value of the Y register passed with this service call. On operating systems supporting the OSRDRM call the Y register contains a negative value, while other versions of the operating system make this call with a positive value in the Y register.

The example given at the end of this section shows how the service routine at the head of a *ROM filing system ROM detects the operating system type and responds appropriately. This example will function on both types of operating system but will take advantage of OSRDRM routine if available. *ROM filing system ROMs designed for use on the earlier operating systems will still work with later versions.

17.5.6 *ROM data format

The format in which data should be stored in *ROM filing system ROMs is very similar to the tape data format. The data is divided into blocks which may be up to 255 bytes long. Each block of data is preceded by a header which contains information about the block. Both the block of data itself and the header are followed by a 16 bit cyclic redundancy check (CRC) value. The filing system calculates its own values for these CRCs during the loading process and compares them. If the filing system's value differs from the stored value then the filing system flags an error and rejects the data. (A routine for calculating CRCs is included in the example at the end of this section.)

The *ROM filing system data format is as follows:

offset	*description*	*length*
	1st BLOCK HEADER	
0	&2A, a synchronisation byte	1
1	file name (1-10 chars.)	n
1+n	&00, a file name terminator	1
2+n	load address (lo byte first)	4
6+n	execution address (lo byte first)	4
10+n	block number (lo byte first)	2
12+n	block length (m, in bytes)	2
14+n	block flag	1
15+n	address of next file	2
17+n	header CRC (1 to 16+n inclusive)	2
	BLOCK DATA	
19+n	data	m
19+n+m	data block CRC	2
	MIDDLE BLOCK HEADER	
prev. blk+1	&23 (#)	1
	LAST BLOCK HEADER	
end	&2b, end of ROM marker	1

bit	meaning
0	*RUN only
1	not used
2	not used
3	not used
4	not used
5	not used
6	no data
7	last block

For the *ROM filing system the headers for all but the first and last blocks may be replaced by a single byte header of value &43 ('#') with no CRC. This is implemented to reduce the memory overheads inherent with the tape style data format.

By convention the first file in a *ROM filing system ROM should be a title file. This is a file of zero length which serves to identify the ROM. The name of this file will appear on catalogue listings of the ROM. The file name of this title file should consist of a name and a version number preceded and followed by an asterisk e.g.'*Mon00*' or '*GAMES05*'.

17.5.7 An example of a *ROM filing system ROM

The program below is written in BASIC 2 to assemble a ROM image which can be 'blown' into an EPROM and placed in a BBC microcomputer paged ROM socket or into a ROM cartridge slot on the Electron Plus 1 expansion.

Included in the program below is a routine for calculating CRC values (FNdo_crc). The actual CRC values required for this ROM image are included in the comments so that the actual values may be inserted directly if someone wanted to reduce the typing load when trying out this example.

```
 10 REM ***************************************
 20 REM *                                     *
 30 REM *   *ROM filing system ROM example    *
 40 REM *                                     *
 50 REM ***************************************

 60 REM Assemble CRC calculating routine

 70 DIM MC% &100:PROCassm

 80 REM Set up constants required for ROM assembly

 90 serROM=&F5
100 ROMid=&F4
110 ROMptr=&F6
120 OSRDRM=&FFB9
130 version=0

140 REM Reserve space for ROM image and prepare to assemble
150 DIM code% &4000
160 FORI=4 TO 7 STEP3
170 P%=&8000:O%=code%
180 [
190 OPT I
200 .ROMstart EQUB 0                    \ null language entry
210           EQUB 0
220           EQUB 0
230           JMP service               \ service entry point
240           EQUB &82                  \ ROM type, service ROM
250           EQUB copyr-ROMstart       \ offset to copyright$
260           EQUB version              \ binary version number
```

```
270             EQUS "Serial Rom"     \ ROM title string
280             EQUB 0
290             EQUS "0"              \ ROM version string
300 .copyr      EQUB 0
310             EQUS "(C) 1982 Acorn Computers" \ copyright$
320             EQUB 0                \ end of paged ROM header
330 .service    CMP #&D               \ service routine
340             BEQ initsp            \ initialise call?
350             CMP #&E
360             BEQ rdbyte            \ read byte call?
370             RTS                   \ not my call

380 \ Routine for paged ROM service call &D

390 .initsp     PHA                   \ save accumulator
400             JSR invsno            \ invert *ROM number
410             CMP ROMid             \ compare with ROM id
420             BCC exit              \ if *ROM > me, not my call
430             LDA #data AND 255     \ low byte of data address
440             STA ROMptr            \ store in pointer location
450             LDA #data DIV &100    \ high byte of data address
460             STA ROMptr+1          \ store in pointer location
470             LDA ROMid             \ get my paged ROM number
480             JSR invert            \ invert it
490             STA serROM            \ make me current *ROM
500 .claim      PLA                   \ restore accumulator/stack
510             LDA #0                \ service call claimed
520             RTS                   \ finished
530 .exit       PLA                   \ call not claimed restore
540             RTS                   \    accumulator and return

550 \ Routine for paged ROM service call &E

560 .rdbyte     PHA                   \ save accumulator
570             TYA                   \ copy Y to A
580             BMI os120             \ if Y -ve  OS has OSRDRM

590 \ this part for OS with no OSRDRM

600             JSR invsno            \ invert *ROM number
610             CMP ROMid             \ is it my paged ROM no.
620             BNE exit              \ if not do not claim call
630             LDY #0                \ Y=0
640             LDA (ROMptr),Y        \ load A with byte
650             TAY                   \ copy A to Y
660 .claim1     INC ROMptr            \ increment ptr low byte
670             BNE claim             \ no overflow
680             INC ROMptr+1          \ increment ptr high byte
690             JMP claim             \ claim call and return

700 \ this part for OS with OSRDRM

710 .os120      JSR invsno            \ A=current *ROM number
720                                   \ not necessarily me
730             TAY                   \ copy A to Y
740             JSR OSRDRM            \ OS will select ROM
750             TAY                   \ byte returned in A
760             JMP claim1            \ incremnt ptr & claim call

770 \ Subroutine for inverting *ROM numbers
```

```
780 .invsno   LDA serROM                \ A=*ROM number
790 .invert   EOR #&FF                  \ invert bits
800           AND #&F                   \ mask out unwanted bits
810           RTS                       \ finished

820 \ End of header code/beginning of data

830 .data     EQUB &2A                  \ synchronisation byte
840 .hdstrt   EQUS "*EXAMPLE*"          \ *ROM title
850           EQUB 0                    \ name terminator
860           EQUD 0                    \ load address=0
870           EQUD 0                    \ execution address=0
880           EQUW 0                    \ block number=0
890           EQUW 0                    \ block length=0
900           EQUB &C0                  \ block flag
910           EQUD eof                  \ pointer to next file
920 .hdcrc    EQUW FNdo_crc(hdstrt,hdcrc) \ CRC (&246F)
930 .eof

940 \ No data block for this file

950           EQUB &2A                  \ synchronisation byte
960 .file1    EQUS "TEXT"               \ file title
970           EQUB 0
980           EQUD 0                    \ null load address
990           EQUD 0                    \ null execution address
1000          EQUW 0                    \ first block
1010          EQUW dat2-dat1            \ length of data
1020          EQUB &80                  \ first & last block
1030          EQUD eof1                 \ pointer to end of file
1040 .hdcrc1  EQUW FNdo_crc(file1,hdcrc1) \ CRC (&E893)
1050 .dat1    EQUS "REM This is a very short text file."
1060          EQUB &D                     \ The file contents !
1070 .dat2    EQUW FNdo_crc(dat1,dat2) \ Block CRC (&655D)
1080 .eof1
1090          EQUB &2B                  \ end of ROM marker
1100 .eor
1110 ]
1120 NEXT
1130 PRINT" *S.ROM ";~code%;" ";~O%
1140 END

1150 REM Define function which calculates CRC
1160 REM Requires start and end of block upto 255 bytes

1170 DEF FNdo_crc(start,end)
1180 ?&82=(start-&8000+code%) AND &FF
1190 ?&83=(start-&8000+code%) DIV &100
1200 ?&84=end-start
1210 CALL crc
1220  =(!&80) AND &FFFF

1230 REM Define procedure which assembles CRC routine

1240 DEF PROCassm
1250 start_addr=&82
1260 Lo_crc=&81
1270 Hi_crc=&80
1280 len=&84
1290 FORI=0 TO 3 STEP3
1300 P%=MC%
```

```
1310 [
1320           OPT I
1330 .crc      LDA #0
1340           STA Hi_crc
1350           STA Lo_crc
1360           TAY
1370 .label1   LDA Hi_crc
1380           EOR (start_addr),Y
1390           STA Hi_crc
1400           LDX #8
1410 .label2   LDA Hi_crc
1420           ROL A
1430           BCC label3
1440           LDA Hi_crc
1450           EOR #8
1460           STA Hi_crc
1470           LDA Lo_crc
1480           EOR #&10
1490           STA Lo_crc
1500 .label3   ROL Lo_crc
1510           ROL Hi_crc
1520           DEX
1530           BNE label2
1540           INY
1550           CPY len
1560           BNE label1
1570           RTS
1580 ]
1590 NEXT
1600 CALL crc:ENDPROC
```

When the resultant ROM is installed in the machine the following dialogue may ensue.

```
>*ROM
>*CAT
*EXAMPLE*
TEXT
>*EXEC TEXT
>REM This is a very short text file.
```

17.6 Paged ROM associated routines

The calls listed here have miscellaneous functions in relation to these ROMs.

17.6.1 OSRDRM, read byte from paged ROM routine

Call address &FFB9
No indirection address

This routine is implemented on the BBC microcomputer and the Electron but is not available in the Tube operating system.

This call returns a byte read from a paged ROM.

Entry parameters:
ROM number stored in Y.
Address stored in &F6 and &F7.

On exit:
A contains the value of the byte read.

17.6.2 Enter language ROM OSBYTE call

Call address &FFF4
Indirected through &20A
A=&8E (142)

Entry parameter:
X determines which paged ROM is entered

The selected language will be copied across to a second processor if one is present. The action of this call is to printout the language name and enter the selected language ROM at &8000 (or the language's relocation address if relocated across the Tube) with A=1. Locations &FD and &FE in zero page are set to point to the copyright message in the ROM.

There is no exit from this call.

17.6.3 Issue paged ROM service call OSBYTE call

Call address &FFF4
Indirected through &20A
A=&8F (143)

Issue paged ROM service call

See Paged ROMs section 17.5.1.

Entry parameters:
X=reason code
Y=parameter passed with service call

On exit:
Y may contain return argument (if appropriate)

X=0 if a paged ROM claimed the service call

A is preserved
C is undefined

17.6.4 Read address of ROM pointer table OSBYTE calls

Call address &FFF4
Indirected through &20A
A=&A8 (168) - low byte of address
A=&A9 (169) - high byte of address

When used across the Tube the address returned refers to the i/o processor's memory.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This table of extended vectors consists of 3 byte vectors in the form Location (2 bytes), ROM no. (1 byte). See Paged ROM section 17.4.3 for a complete description of extended vectors.

On exit:

X=low byte (if called with A=&A8)
Y=high byte (if called with A=&A8)

A is preserved
C is undefined

17.6.5 Read address of ROM info table OSBYTE calls

Call address &FFF4
Indirected through &20A
A=&AA (170) - low byte of address
A=&AB (171) - high byte of address

This call is implemented on the BBC microcomputer and the Electron. When used across the Tube the address returned refers to the i/o processor's memory.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. the contents of the next location are returned in Y.

This call returns the origin of a 16 byte table, containing one byte per paged ROM. This byte contains the ROM type byte contained in offset &06 of the ROM or contains 0 if a valid ROM is not present.

On exit:

X=low byte (if called with A=&AA)
Y=high byte (if called with A=&AA)
A is preserved
C is undefined

17.6.6 Read BASIC ROM number OSBYTE call

Call address &FFF4
Indirected through &20A
A=&BB (187)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old ROM number is returned in X.

Basic is recognised by the fact that it is a language ROM which does not possess a service entry. This ROM is then selected by the *BASIC command. If no BASIC ROM is present then this location contains &FF.

17.6.7 Read current language ROM number OSBYTE call

Call address &FFF4
Indirection address &20A
A=&FC (252)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old ROM number is returned in X.

This location is set after use of OSBYTE &8E/*FX 126. This ROM is entered following a soft BREAK or a BRK (error).

17.6.8 100Hz paged ROM polling system

On the Electron and Master series computers a mechanism has been implemented to provide ROMs with a paged ROM service call at centisecond intervals. This polling call with A=&15 is described in section 17.5.1. The following calls are used to manipulate the polling semaphore.

17.6.8.1 Increment polling semaphore OSBYTE call

Call address &FFF4
Indirection address &20A
A=&16 (24)

This call is only implemented on the Electron and Master series computers.

This call increments the semaphore which when non-zero makes the operating system issue a paged ROM service call with A=&15 at centi-second intervals.

On exit:

A and X are preserved
Y and C are undefined

17.6.8.2 Decrement polling semaphore OSBYTE call

Call address &FFF4
Indirection address &20A
A=&17 (25)

This call is only implemented on the Electron and Master series computers.

This call decrements the semaphore which when non-zero makes the operating system issue a paged ROM service call with A=&15 at centi-second intervals.

On exit:

A and X are preserved
Y and C are undefined

17.6.8.3 Read/write polling semaphore OSBYTE call

Call address &FFF4
Indirected through &20A
A=&B9 (185)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old semaphore value is returned in X.

This location contains the semaphore. This semaphore should only be read using this call.

17.6.9 Check for 6502 code OSBYTE

Call address &FFF4
Indirected through &20A
A=&A4 (164)

On entry:

X and Y contain the address of the code to be checked.

This call first checks that the code is in a paged ROM format by looking for the copyright string at offset &07. It then checks the processor type byte at offset &06. If bit 7 is clear a BRK is generated with the error string 'This is not a language'. If the processor type bits (bits 0-3) indicate that the ROM is not 6502 code or 6502 BASIC then a BRK is executed with the error message 'I cannot run this code'.

18 Second processors and the Tube™

The BBC Acorn range of machines are designed to interface to a second processor via a hardware device/software protocol called the Tube. A number of different microprocessors may be used as second processors such as the 6502, the Z80, the NS 16032 and the Acorn RISC Machine. The Tube provides an interface between two microcomputers allowing data to be exchanged at a clocked rate of 2MHz.

Each second processor is a very basic microcomputer consisting of a microprocessor, some RAM, an operating system ROM and a Tube ULA. All its input and output (i/o) eminates from a host processor such as the BBC microcomputer. When a second processor is powered up it immediately starts looking at the registers in its Tube ULA; it listens for instructions from its host or i/o processor. The second processor may be called a parasite processor because it is totally reliant on its i/o processor for all input or output.

By using a different microprocessor as the second processor it is possible to considerably increase the flexibility of the microcomputer system. A system comprising of a BBC microcomputer and a Z80 second processor is able run a vast range of CP/M software in addition to the software written to run on the BBC micro's own 6502. The Master 512 provides an 80186 second processor and emulates the IBM PC.

18.1 Tube system 32 bit addressing

The relevance of the 32 bit addressing scheme used by filing systems and the 'machine high order address' returned by OSBYTE &82 become apparent in the context of the Tube. The 32 bit address defined by Acorn is designed to provide a 4 Gigabyte logical address space which includes the memory of both the i/o processor (the host BBC microcomputer) and the second processor.

Addresses in the region &FFFF0000 to &FFFFFFFF (the top 64Kb of the 4 Gb) refer to the i/o processor memory map leaving the rest of the logical address space for second processor memory. When filing systems fail to detect the presence of a second processor then only the least significant two bytes of the address are used. These two bytes are used to refer to the 64K of addressable memory in the Acorn BBC microcomputer series machine.

18.2 OS calls made from second processors

The 6502 based second processors provide almost identical facilities for executing operating system calls to those present on normal BBC microcomputers. Clearly this is important so as to allow programs which run on unexpanded machines to run on second processors.

Non-6502 based second processors clearly have no requirement to be identical to the host environment and clearly this would be difficult to achieve since the register names and functions are different on different microprocessors. However Acorn have made every attempt to provide the same operating system facilites available on different second processors. These equivalent calls have been designed to make them as similar as possible. Thus on the Z80 second processor the OSBYTE and OSWORD call addresses remain &FFF4 and &FFF1 respectively with the accumulator value selecting the routine but the H and L registers of the Z80 are used in place of the X and Y registers of the 6502.

Most second processors run a proprietary operating system of their own which provide calls which provide the usual i/o facilities.

18.3 The Tube ULA

The Tube ULA is a custom chip designed by Acorn to allow extremely rapid communication between the second processor and its i/o processor. To each of the two processors the Tube resembles a conventional peripheral device which occupies 8 bytes of memory in the i/o space. There are four, 8 bit wide read-only ports and four 8 bit wide write-only ports with their associated control registers.

The following table gives a brief description of the Tube registers. This amount of detail is excessive for most users and is included to indicate the general design of the tube rather than to provide a reference for low level tube use.

Tube Register	name	host address	parasite address (6502)
Status Register 1	**R1STAT**	**&FEE0**	**&FEF8**
	bit 7	DA1	data present in 1st data register
	bit 6	NF1	1st data register not filled
	bit 5	P	set parasite reset active low
	bit 4	V	enable 2 byte fifo use of 3rd data register
	bit 3	M	enable parasite NMI from 3rd data register
	bit 2	J	enable parasite IRQ from 4th data register
	bit 1	I	enable parasite IRQ from 1st data register
	bit 0	Q	enable host IRQ from 4th data register
Data Register 1	**R1DATA**	**&FEE1**	**&FEF9**
	used for OSWRCH, events and ESCAPE handling writing to this register generates an IRQ in the co-processor		
Status Register 2	**R2STAT**	**&FEE2**	**&FEFA**
	bit 7	DA2	data present in 2nd data register
	bit 6	NF2	2nd data register not filled
	bits 5-0		not used
Data Register 2	**R2DATA**	**&FEE3**	**&FEFB**
	used by other operating system calls		
Status Register 3	**R3STAT**	**&FEE4**	**&FEFC**
	bit 7	DA2	data present in 3rd data register
	bit 6	NF2	3rd data register not filled
	bits 5-0		not used
Data Register 3	**R3DATA**	**&FEE5**	**&FEFD**
	user register for data transfer and i/o errors writing to this register generates an NMI in the co-processor		
Status Register 4	**R4STAT**	**&FEE6**	**&FEFE**
	bit 7	DA2	data present in 4th data register
	bit 6	NF2	4th data register not filled
	bits 5-0		not used
Data Register 4	**R4DATA**	**&FEE7**	**&FEFF**
	system register used for control of data transfer writing to this register generates an IRQ in the co-processor		

The byte-wide communication channels fall into two distinct categories. The first are simply latches, so data written to them on one side can be read directly at the other. The second set are FIFO (first in first out) buffers allowing the temporary storage of data within the Tube ULA in the same way as the printer buffer temporarily stores the characters for the printer. The FIFO buffered channels enable the second processor to queue certain operating system call requests in a limited way.

The user will have no need to interfere with the Tube ULA directly except when implementing custom filing systems (see section 18.6). The operating system on the second processor will allow the use of all the facilities on the i/o processor by duplicating any appropriate i/o processor OS call in the second processor OS.

18.4 The Tube software on the i/o processor

The software which services any i/o requests from the i/o processor presented via the Tube assumes the *current-language* status in the i/o processor. The DNFS ROM will recognise an active Tube during a reset and copy the Tube servicing routines from within itself down into the language workspace. The Tube servicing routines are not executed from within a paged ROM to reduce unnecessary switching between ROMs. The i/o processor and the Tube servicing code never knows the identity of the processor on the other side of the Tube; requests presented via the Tube are handled in the same manner for each of the second processor types.

During a reset or following a language re-selection, the selected language is copied across the Tube and execution in the second processor is attempted. This process can be considered like a *RUN request to a filing system and indeed language ROMs may be stored on disc and executed in exactly this way on a second processor. It is up to the user to ensure that the language or file selected for running on a second processor is appropriate for that processor type. Language ROMs should contain processor type information (see Paged ROMs section 17.1.3) to enable a second processor to recognise languages not intended for it. It is also possible to provide a re-location address in language ROMs so that the language is copied to a specific address in the second processor's memory (see section 17.1.9).

The Tube code starting at &400 in the i/o processor contains a number of entry points used by the operating system. There is only one entry point which may be used by other software such as filing systems; this is described in section 18.6. The Tube code entry points are described in the following table.

&400	Copy language to second processor
&403	Copy ESCAPE flag to second processor
&406	Transfer data between host and second processor

18.5 The 6502 as a typical second processor

The 6502 second processor is described below in detail. The use of other second processors requires the same considerations but they are unlikely to be used in a similar manner. Many problems are encountered by users of the 6502 second processor because they are not fully aware that their programs are running on a different physical microprocessor. Users of other microprocessor types rarely need reminding of this fact but they will occasionally require an understanding of the nature of the relationship between the second processor and the i/o processor.

When a 6502 second processor is active on a BBC microcomputer system the user should not be aware of the fact his programs are now running on a different microprocessor. While this statement is largely true for BASIC programmers, word processors and the like, it is not entirely true for those programming in machine code. Acorn computers have constantly emphasized the importance of using official operating system routines in software written for use on the BBC Acorn range of machines. Once a knowledge and understanding of these has been gained the advanced programmer can take every advantage of the extra facilities available on a second processor while writing code which also functions effectively on an unexpanded BBC Microcomputer or Electron.

While the 6502 second processor provides the user with increased user RAM and a faster microprocessor it is not just a composite RAM and turbo-charger expansion card. An extra 64K of memory is added to the system but all this memory is not available as a contiguous block on top of the memory on an non-Tube machine. The second processor in this case is a separate microcomputer which offers an identical programming environment to that of other Acorn BBC range machines. Within the second processor a miniscule amount of RAM is used by the operating system thus an increased amount of user RAM is available (up to 44K with BASIC relocated to &B800). Meanwhile the user RAM on the i/o processor is released for the use of an exploded font (default state on a Tube machine, see OSBYTE &14), service type paged ROMs and miscellaneous independent user routines (e.g. event handling routines, see section 17.4.1).

The reason that Acorn stress the importance of using official operating system routines for all i/o is because there is a limit to the mimicry of the BBC microcomputers features that can be copied onto second processors. The Tube operating system cannot distinguish between different types of memory use. Software which assumes it is running on a non-Tube machine and pokes screen memory directly, will not have the desired effect when running on a second processor.

The systems programmer should bear in mind the information laid out in this chapter when writing software which will be required to work across the Tube. A microcomputer with an active second processor is referred to as an i/o processor.

18.6 Using OS calls and vectors

The operating system in the 6502 second processor implements most of the OS routines available on the i/o processor as well as the vectors in page &02. Most of these second processor operating system routines involve passing the request across the Tube to the i/o processor where the operating system proper performs the required function.

Each routine such as OSRDCH, OSWRCH etc. is implemented in the second processor's operating system at exactly the same address as on the i/o processor's operating system. Where the equivalent i/o processor's OS routine is indirected through a vector the second processor's OS indirects its routine through a vector at the same address.

The dialogue which is invoked across the Tube following operating system calls is described in the following table. The operating system in the second processor initiates communication with the host i/o processor by writing a reason code to one of the tube registers which defines which service is required.The Tube hardware causes an interrupt to be generated in the i/o processor which then immediately services the request from the second processor. Indiscriminate writing to the Tube registers can crash the i/o processor by causing unserviceable interrupts.

This table describes the data register set used by operating system calls and the parameters exchanged across the tube as the call is invoked and returns.

Call name	Call Parameters: reason code	Call Parameters: data reg.	Call Parameters: no. of add. bytes	Call Parameters	Return Parameters: no. of bytes returned	Return Parameters
OSRDCH	2	&00	0		2	C flag (bit 7), then character
OSCLI	2	&02	var.	command line terminated by &0D	1	&7F when completed
OSBYTE (A<&80)	2	&04	2	X register then Acc. values	1	X register result
OSBYTE (A>&7F)	2	&06	3	X, Y and Accumulator values	3	carry (bit 7), Y and X results
OSWORD (A<>0)	2	&08	var.	no. of bytes, then data	var.	no. of bytes, then data
OSWORD (A=0)	2	&0A	5	max, min & length, buffer addr.	var.	&FF if ESC or &7F ack. then line
OSARGS	2	&0C	6	Y reg, 4 bytes, Accumulator	5	A register result, 4 bytes returned
OSBGET	2	&0E	1	Y register value	2	C flag (bit 7), then Accum. result
OSBPUT	2	&10	2	Y register then Acc. values	1	&7F when completed
OSFIND	2	&12	var.	Acc. then Y reg or filename	1	filehandle or &7F
OSFILE	2	&14	17	bytes 3-18 of block, Accum.	17	Acc. then returned param. block
OSGBPB	2	&16	14	OSGBPB params, A value	15	param. block, C flag, Accum.
OSWRCH	1	n/a	1	character	0	

Calls made from the second processor and passed to the i/o processor

host BRK	4	&FF then via reg. 2: &00 - BRK, error code, error message terminated by &00
events	1	&00, Y reg value, X reg value, Accumulator value
ESC flag change	1	one byte: bit 7=1, bit 6=new flag
begin data transfer	4	&406 entry value, claim id number, 4 byte address (hi byte 1st), reset value

Calls originating in the i/o processor which require action form the second processor

It should be noted however that operating system calls originating in the i/o processor will not be offered to the vector on the second processor. This means that filing system messages for example cannot be intercepted by the OSWRCH vector on the second processor, output originating from the current language, however, will be offered via this vector (the current language is copied across to the second processor).

The description of the operating system calls in chapter 6 includes individual comments on the use of these calls over the tube. In addition a few relevant points are made in the following sections.

18.6.1 Events

Events are passed to the event vector on the second processor in exactly the same manner as they are passed to the event vector in the i/o processor (see section 7.1). In addition to the rule that event handling routines should not last more than 10ms on second processors, event handling routines should not use operating system routines. It should be

remembered that event handling routines may still use operating system routines on a Tube machine if the event handling code is resident in the i/o processor. The example below shows how a program resident in a 6502 second processor can place an event handling routine in the i/o processor.

This program sets up an event handling routine which causes a short sound to be emitted in response to each key press. The frequency of the sound varies according to the key value.

```
 10 OSWORD=&FFF1
 20 OSBYTE=&FFF4
 30 EVNTV=&220

 40REM*******
 50REM        Read OSHWM on i/o processor
 60REM*******

 70 A%=180:X%=0:Y%=&FF
 80 OSHWM=(USR OSBYTE) AND &FF00
 90 DIM MC% 100,X% 8:Y%=X% DIV 256
100 FOR opt%=4 TO 7 STEP3
110   P%=OSHWM
120   O%=MC%
130   [
140         OPT opt%
150.entry   PHP
160         PHA
170         TXA
180         PHA
190         TYA
200         PHA                   \ save registers
210         STY sound+4           \ SOUND pitch=key ASCII value
220         LDX #sound AND 255
230         LDY #sound DIV 256
240         LDA #7
250         JSR OSWORD            \ perform SOUND command
260         PLA
270         TAY
280         PLA
290         TAX
300         PLA
310         PLP                   \ restore registers
320         RTS                   \ return from event handler
330.sound   EQUD &FFF50001        \ set up SOUND 1,-11,x,1
340         EQUD &00010000
350   ]
360   NEXT opt%

370REM*******
380REM        Copy routine over to i/o processor
390REM*******

400 I=0
410 REPEAT
420  PROCWRITEIO(I?MC%,OSHWM+I)
430  I=I+1
```

```
440  UNTIL I+OSHWM=P%

450REM*******
460REM        Set up EVNTV in i/o processor
470REM*******

480 PROCWRITEIO(entry AND 255,EVNTV)
490 PROCWRITEIO(entry DIV 256,EVNTV+1)

500REM*******
510REM        Enable keyboard event
520REM*******

530 *FX14,2
540 END

550REM*******
560REM          Procedure for writing byte to i/o processor
570REM*******

580 DEF PROCWRITEIO(data,addr):!X%=addr:X%?4=data
590  A%=6:CALL OSWORD:ENDPROC
```

An alternative method of loading and initialising this event handling routine would be to include an initialising routine in the machine code part of the program and *SAVE a copy of the routine using a high order address specifying the i/o processor (precede addresses with &FFFF) for the execution and reloading addresses. *RUNning this file would then cause it to be loaded and executed within the i/o processor. (For an example of this look at the example in the introduction of chapter 6.)

18.6.2 Interrupts

Interrupts from the i/o processor are not passed to the second processor and so the vectors IRQ1V and IRQ2V are not implemented by the second processor's operating system. Thus interrupts generated by the i/o processor hardware can only be trapped by code on the i/o processor. Interrupt handling routines should be implemented in service type paged ROMs where possible. When a user wants to implement an interrupt handling routine resident in i/o processor RAM (like the event handling routine above) the problem arises as to how to change the interrupt vector in the i/o processor. The interrupt vectors should not be changed while interrupts are enabled and it is not possible to disable interrupts on the i/o processor from a second processor while the vectors are being written. A way round this would be to design an event handling routine running in the i/o processor which initialises the interrupt handling routine. Event number 4 (start of vertical sync) will occur within 0.02s of being enabled. The event handling routine can disable the event and restore the vector after initiating the interrupt handling routine. Alternatively the interrupt handling initialisation can

be performed by *RUNning a program which has i/o processor addresses as described above.

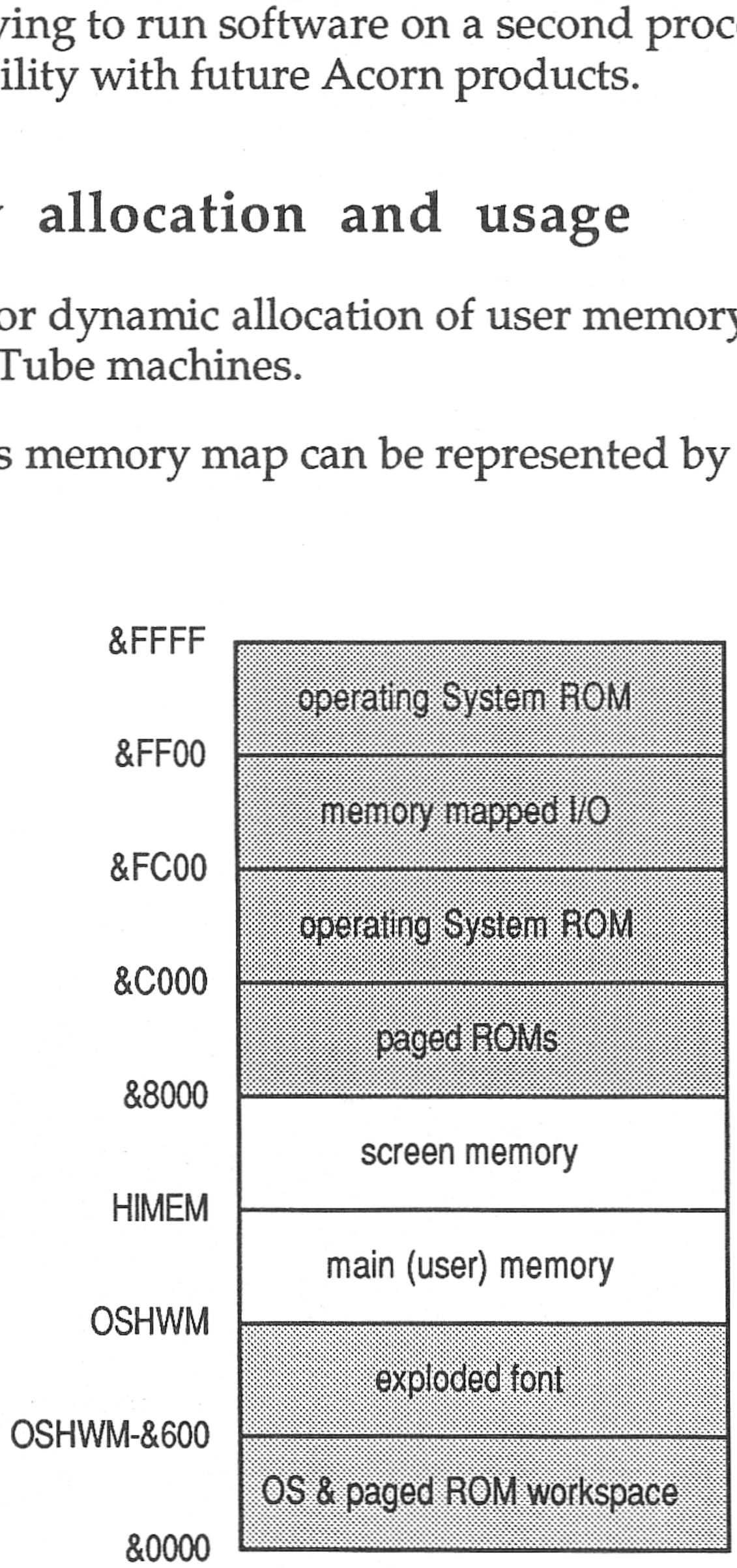
18.6.3 OSBYTE calls

Calls &00 to &7F only return a value in the X register. The high number calls return the X and Y registers and the carry flag. On a non-Tube machine some of these calls return information beyond these limits. Acorn would like to stress that this information is officially undefined and not supported. The non-standard use of these calls may lead to problems when trying to run software on a second processor and will lead to incompatibility with future Acorn products.

18.7 Memory allocation and usage

The mechanisms for dynamic allocation of user memory are the same for Tube and non-Tube machines.

The i/o processor's memory map can be represented by the diagram below.

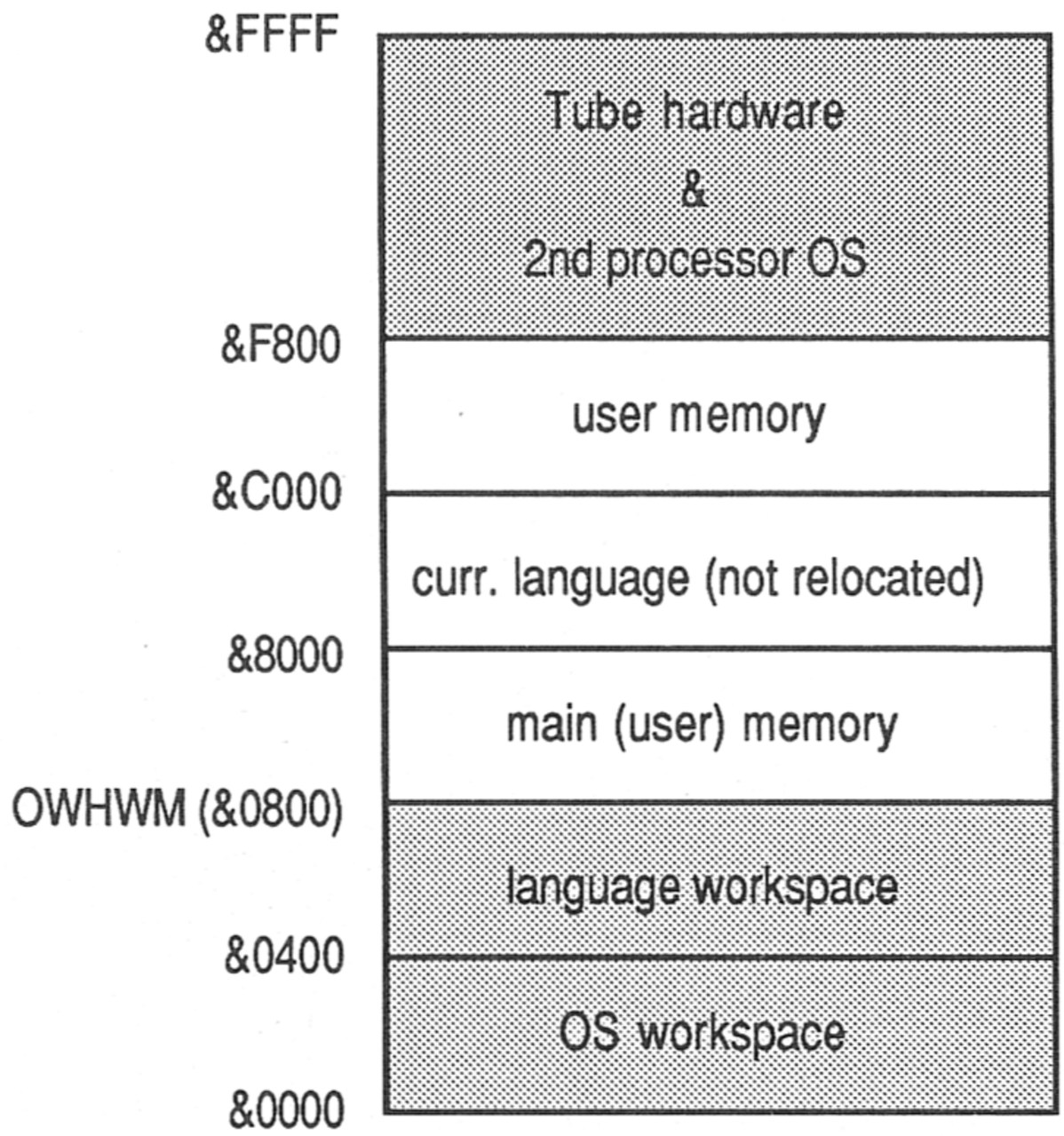


This memory map is essentially unchanged from the memory map of a non-Tube machine. There are two aspects which should be noted by

programmers wanting to use the user memory on the i/o processor. The default state of the font RAM allocation is such that it is fully exploded when a Tube is active (see OSBYTE &14 for more information about font explosion). This means that an extra &5FF bytes are taken up by the operating system workspace. The OSHWM in the i/o processor may be read using OSBYTE &B4. This value will be &600 greater than the value when the Tube is inactive on the same machine e.g. &2100 on a BBC microcomputer with DFS only (please note that programs should ALWAYS interrogate the machine for the current value of OSHWM i.e. do not assume a fixed value for it when writing programs.)

The Tube software in the i/o processor is allocated the memory which would otherwise be used by the current language on a non-Tube machine. The DNFS ROM actually copies down the Tube software to the language workspace (pages 4 to 7) in the i/o processor.

The memory map on the second processor is described in the diagram below.



Zero page is available for users up to location &EE. Page 2 locations not used for indirection vectors by the second processor OS are also available to users.

The language workspace (pages 4 to 7) may be used by the user when not running a language.

18.8 Protocol for transferring data across the Tube

Software running in the i/o processor may require to transfer data to and from the second processor. Filing systems will obviously need to do this. OSWORDs &05 and &06 may be used for small amounts of data (see section 18.9) otherwise the following protocols should be used.

Routines are implemented within the Tube handling software in the i/o processor to enable this to be done. These routines must be used in accordance to a strict protocol if they are to function correctly.

The paragraphs below describe how software may use the Tube.

(a) Claiming the Tube

Before a piece of software can use the Tube it must first claim it successfully. A claim will not necessarily be immediately successful because there may be a number of other processes wanting to use the Tube at the same time. The Tube software has to overcome the problems of background and foreground tasks clashing. The type of problem which might occur is an attempt by interrupt code to use the same routine that the interrupted code was using at the time of the interrupt. These problems occur because not all of the Tube subroutines are re-entrant and interrupts cannot always be disabled.

Before attempting to use any of the Tube routines an OSBYTE call with A=&EA, X=0 and Y=&FF should be made to establish whether a Tube is present on the machine. The X register will be returned with the value &FF if a Tube is present and with zero otherwise. When this call has confirmed the presence of a Tube, the Tube code entry point may be used; if this entry point is used when no Tube is present you will be attempting to execute the language workspace.

To claim the Tube a call is made to the Tube code entry point at &406. This code should be entered with a reason code and a caller's ID in the accumulator. The reason code for this call is &C0 to which should be added a 6 bit ID code which identifies the caller uniquely. This call returns with the carry flag set if the claim was successful or with the carry flag clear it failed. The calling routine should continue to attempt to claim the Tube until successful. The Tube will be re-allocated when the background task has completed its transaction.

A typical claiming routine might be:

```
.claim      PHA
.re_try     LDA #&C1
            JSR &0406
            BCC re_try
            PLA
            RTS
```

Some caller IDs have been allocated to filing systems. They are:

&0	Cassette filing system
&1	Disc filing system
&2	Econet filing system (low level primitives)
&3	Econet filing system (high level primitives)

The actual ID is a 6 bit value which should be added to the reason code giving the accumulator value to be used for claiming and relinquishing Tube ownership.

(b) Relinquishing the Tube

When a routine has finished with the Tube it must release the Tube so that other users may claim it. The use of the caller ID prevents software which did not originally claim the Tube from releasing it. A reason code of &80 to which is added the caller's ID should be placed in the accumulator and the Tube code entry point called.

A typical releasing call might be:

```
.release    PHA
            LDA #&81
            JSR &0406
            PLA
            RTS
```

(c) Data transfer or Execution

To transfer data and execute code in the second processor the Tube code entry point is called with accumulator reason codes in the range 0 to 7. The X and Y registers should contain values pointing to a parameter block in the i/o processor's memory. This parameter block should contain a 16 bit address in the second processor. Depending on the reason for the call the address passed to the routine indicates an address in the second processor from which or to which data should be transferred or an execution address. These addresses should be stored as low byte - high byte.

When using calls for the transfer of data the Tube system requires to be allowed a certain response time before being called again. Therefore the protocol requires that there is an initial delay after using the calls described below and susbsequently a delay after reading each byte or pair of bytes before attempting to read the next byte or pair of bytes.

The actual bytes transferred are read from or written to location &FEE5. This i/o processor address represents the address at which one of the Tube ports is wired. The read or write operation is initiated by making a call to the Tube code entry point with the appropriate entry parameters. After control returns to the calling routine there should be a pause to cover the initial delay before the first read or write to the Tube port. Thereafter there should be a pause between each subsequent read or write. Unless there is an implicit release of the Tube within the call definition, the calling routine should explicitly release the Tube when the transfer is complete.

Call address = &406

A = reason code
X = least significant byte of parameter block address
Y = most significant byte of parameter block address

Parameter block:

XY +	0	second processor address LSB
	1	
	2	
	3	second processor address MSB

Reason Code Summary:

reason code	description	from	to	init delay µs	loop delay µs
0	multiple single byte transfer	2nd	I/O	24	24
1	multiple single byte transfer	I/O	2nd	0	24
2	multiple double byte transfer	2nd	I/O	26	26
3	multiple double byte transfer	I/O	2nd	0	26
4	execute code in 2nd processor				
5	reserved				
6	256 byte transfer	2nd	I/O	19	10*
7	256 byte transfer	I/O	2nd	0	10*

(* for each byte transferred)

Reason code 0
Multiple single byte transfer: second processor to i/o processor.
Initial delay 24 µS
Transfer delay 24 µS per byte read

This call may be used to read any number of bytes from the second processor. Repeated reads of &FEE5 yield a sequence of bytes read from the second processor's memory starting at the address specified in the parameter block.

Reading should be terminated by releasing the Tube or selecting another mode of operation.

Reason code 1
Multiple single byte transfer: i/o processor to second processor
No initial delay
Transfer delay 24 µS per byte written

This call may be used to write any number of bytes to the second processor. Repeated writes to &FEE5 place a sequence of bytes in from the second processor's memory starting at the address specified in the parameter block.

Writing should be terminated by releasing the Tube or selecting another mode of operation.

Reason code 2
Multiple double byte transfer: second processor to i/o processor
Initial delay 26 µS
Transfer delay 26 µS per byte pair read

This call provides a faster protocol for the reading of pairs of bytes from the second processor.

Reading should be terminated by releasing the Tube or selecting another mode of operation.

Reason code 3
Multiple double byte transfer: i/o processor to second processor
No initial delay
Transfer delay 26 µS per byte pair written

This call provides a faster protocol for the writing of pairs of bytes to the second processor.

Writing should be terminated by releasing the Tube or selecting another mode of operation.

Reason code 4
Execute code in the second processor

This call forces execution of code in the second processor at the address contained in the parameter block.

This call contains an implied release and does not return to the caller.

Reason code 5
Reserved for use by operating system calls

Reason code 6
256 byte transfer: second processor to i/o processor
Initial delay 19 µS
Transfer delay 10 µS per byte read

This call provides a very fast protocol for reading 256 bytes. This call functions in the same way as the call with reason code 0 but transfers exactly and only 256 bytes.

After the 256 bytes have been read the Tube should be released or another mode of operation should be selected.

Reason code 7
256 byte transfer: i/o processor to second processor
No initial delay
Transfer delay 10 µS per byte written

This call provides a very fast protocol for writing 256 bytes to the second processor. The call functions in the same way as the call with reason code 1 but transfers exactly and only 256 bytes.

After 256 bytes have been written the Tube should be released or another mode of operation selected.

Below is an example of an assembly language routine which reads a page of the second processor's memory into the i/o processor's memory.

Page zero locations &80 and &81 should be set to point to the destination page in the i/o processor.

Locations &3000, &3001, &3002 and &3003 contain the source address in the second processor.

```
.claim      LDA #&C0+&10 \ caller ID=&10
            JSR &406     \ claim Tube
            BCC claim

            LDX #0       \ low byte of parameter block address
            LDY #&30     \ high byte of parameter block address
            LDA #6       \ reason code for 256 byte read
            JSR &406     \ make call

            LDX #6       \ delay loop 37 machine cycles
.wait       DEX          \ at 2 MHz this is 18.5 microseconds
            BNE wait

            LDY #0
.loop       LDA &FEE5    \ read byte from port
            STA (&80),Y  \ store byte
            NOP
            NOP          \ delay, loop total 10.5 microseconds
            NOP
            INY          \ increment loop counter
            BNE loop     \ get next byte

            LDA #&80+&10 \ release code + caller ID
            JSR &406     \ relinquish Tube
.end
```

This routine has been loosely incorporated into the next example. The program assembles a paged ROM which responds to the command *TUBE by copying 20 Kb from second processor memory into the I/O processor's memory. The program fills the IO processor's screen memory with data from memory at the same address in the second processor.

The program uses two bytes of zero page workspace (`io_ptr`, `io_ptr+1`) as a pointer into I/O processor memory, and four bytes of main memory as a parameter block for the tube call. The variables `io_address` and `sp_address` define the target address in the I/O processor and the source address in the second processor respectively. The variable `pages` defines the number of pages of memory to be transfered.

The timing delay loops have been set up to give the minimum delay on the fastest microprocessor in a BBC series micro (i.e. 3 MHz). It is worth reducing the delay loops when using a slower processor.

This example has been presented as a paged ROM but it would be possible to place a program in the I/O processor and execute it from the second processor (using, for example, the *LINE or *CODE facilities via the user vector). The most natural way of implementing this code would be to incorporate two new OSWORDs in a paged ROM. One OSWORD to provide rapid data transfer from the second processor to the I/O processor and the other to provide the reciprocal service.

```
  1 REM *** Tube Transfer ROM
  2 REM *** an example of data transfer across the Tube
  3 REM *** responds to '*TUBE'
  4 REM ***
 10 DIM code% &400
 20 line=&F2
 30 OSASCI=&FFE3
 40 OSBYTE=&FFF4
 50 io_ptr=&80
 60 par_blk=&2F00
 70 io_address=&3000:pages=&50
 80 sp_address=&3000
 90 id_code=&0F
100 claim_code=&C0
110 release_code=&80
120 read_256=6
130  FOR I=4 TO 7 STEP 3
140    P%=&8000:O%=code%
150    [
160    OPT I
170   .romstrt EQUB 0                    \ no language entry
180            EQUB 0
190            EQUB 0
200            JMP service              \ service entry
210            EQUB &82                 \ ROM type flag
220            EQUB (copyr-romstrt) \ copyright message offset
230            EQUB &00
240   .title   EQUS "TUBE TRANSFER ROM " \ ROM title string
250            EQUB &0
260            EQUS "0.00"              \ ROM version string
270   .copyr   EQUB 0
280            EQUS "(C)1987 Mark Holmes"  \ copyright string
290            EQUB 0
300   .name    EQUS "EBUT"              \ data for command recognition
310   .service CMP #4                   \ does A=5
320            BEQ command              \ if so process command line
330            CMP #9                   \ does A=9
340            BEQ help                 \ if so process *HELP request
350            RTS                      \ otherwise leave alone
360   .command TYA:PHA:TXA:PHA          \ save registers
370            LDX #4                   \ X=4 (length of name 'TUBE')
380   .loop1   LDA (line),Y             \ A=first letter in comnd. line
390            CMP name-1,X             \ compare A with our name
400            BNE notme                \ if not equal, not our name
410            INY                      \ Y=Y+1 for next char.in comnd.
```

```
420             DEX                     \ X=X-1 for next char. in name
430             BNE loop1               \ if X<>0 then comp. next char.
440             BEQ init                \ otherwise command = 'TUBE'
450 .notme      PLA:TAX:PLA:TAY         \ not for us, restore registers
460             LDA #4                  \ restore reason code
470             RTS                     \ .. and return
480 .help       TYA:PHA:TXA:PHA         \ save registers
490             LDA #13                 \ A=carriage return
500             JSR OSASCI              \ perform CR,LF
510             LDX #0                  \ index register, X=0
520 .loop2      LDA title,X             \ A=1st char. in title string
530             BNE over1               \ if A=0 (end of string) branch
540             LDA #&20                \ A=space character
550 .over1      JSR OSASCI              \ print space
560             INX                     \ X=X+1
570             CPX #(copyr-title)      \ if X hasn't reached cpyr.str.
580             BNE loop2               \       then print next string
590             LDA #13                 \ A=carriage return
600             JSR OSASCI              \ perform CR,LF
610             PLA:TAX:PLA:TAY         \ restore registers
620             LDA #9                  \ restore reason code
630             RTS                     \ return
640 .init       LDA #&EA                \ *FX 234,0,255
650             LDX #&0                 \ is tube active OSBYTE
660             LDY #&FF
670             JSR OSBYTE
680             CPX #0                  \ X=&FF if tube present
690             BEQ notube              \ abort if tube inactive
700             LDA #&0                 \ set up tube call parameters
710             STA par_blk+3
720             STA par_blk+2
730             LDA #sp_address DIV 256
740             STA par_blk+1
750             LDA #sp_address AND 255
760             STA par_blk               \ !par_blk=sp_address
770             LDA #io_address DIV 256
780             STA io_ptr+1              \ ?io_ptr+1=io_address (lo)
790             LDA #io_address AND 255
800             STA io_ptr                \ ?io_ptr=io_address (hi)
810             JSR claim                 \ claim tube
820             JSR readsp                \ read data from 2nd proc.
830             JSR release               \ release tube
840             PLA:TAX:PLA:TAY           \ restore registers
850             LDA #&0                   \ replace reason code
860             RTS                       \ return
870 .claim      PHA                       \ save accumulator
880 .re_try     LDA #claim_code+id_code     \ A=reason code + id code
890             JSR &0406                 \ call tube code
900             BCC re_try                \ loop if claim fails
910             PLA                       \ restore accumulator
920             RTS                       \  .. and return
930 .release    PHA                       \ save accumulator
940             LDA #release_code+id_code \ A=reason code + id code
950             JSR &0406                 \ call tube code
960             PLA                       \ restore accumulator
970             RTS                       \ .. and return
980 .readsp     LDA io_ptr+1              \ A=?io_ptr+1
990             CMP #(io_ptr/256)+pages \ if A=&80 ..
1000            BEQ finish                \  .. then we've finished
1010            LDX #par_blk AND 255    \ X=lo byte of param. block
1020            LDY #par_blk DIV 256    \ Y=hi byte of param. block
1030            LDA #read_256             \ A=reason code
```

```
1040            JSR &406                \ call tube code
1050            LDX #9
1060    .wait   DEX
1070            BNE wait                \ total delay >19 microsecs
1080            LDY #0                  \ index register, Y=0
1090    .loop   LDA &FEE5               \ read byte from tube reg.
1100            STA (io_ptr),Y          \ store byte in memory
1110            NOP:NOP
1120            NOP:NOP
1130            NOP:NOP                 \ delay >10 microseconds
1140            INY                     \ Y=Y+1
1150            BNE loop                \ read next byte if Y>0
1160            CLC                     \ clear carry flag
1170            LDA #1                  \ A=1
1180            ADC io_ptr+1            \ A=A+?io_ptr
1190            STA io_ptr+1            \ ?io_ptr=A
1200            CLC
1210            LDA #1
1220            ADC par_blk+1
1230            STA par_blk+1           \ ?&2F01=?&2F01+1
1240            JMP readsp              \ loop back
1250    .finish RTS                     \ return when finished
1260    .notube LDX #0                  \ print message when no tube
1270    .messlp LDA message,x
1280            BEQ quit
1290            JSR OSASCI
1300            INX
1310            JMP messlp
1320    .quit   PLA:TAX:PLA:TAY         \ restore registers
1330            LDA #0                  \ service call claimed
1340            RTS                     \ .. and return
1350    .message EQUS "Tube not active"+CHR$13+CHR$0
1360     ]
1370  NEXT
1380 OSCLI"*S.TBRM "+STR$~code%+" "+STR$~O%
```

The short listing below tests the Tube Transfer ROM. The nested loops set up a chequer-board pattern in the screen memory when run with the Tube disabled (non-shadow RAM mode also necessary). The message 'Tube not active' should also appear when not used on a second processor. Running the program on the 6502 second processor will result in the screen image being formed in second processor memory before being transferred to the screen memory in the I/O processor.

In order to test the ROM and test program on a 32016 second processor with Bas32 it will be necessary to change the value of sp_address in the ROM listing to &8000 and change the value of block in the listing below to the same value. This is because otherwise the data will over write the basic interpreter itself.

```
10 REM *** This program creates a bit pattern
20 REM *** which is then transferred to the I/O processor's
30 REM *** screen memory using the *TUBE command

40 MODE 0
```

```
50 block=&3000
60 FOR I=block TO block+&4D80 STEP &500
70   FOR J=0 TO &270 STEP 16
80     I!(J)=-1:I!(J+4)=-1:I!(J+8)=0:I!(J+12)=0
90   NEXT
100   FOR J=&280 TO &4F0 STEP 16
110     I!(J)=0:I!(J+4)=0:I!(J+8)=-1:I!(J+12)=-1
120   NEXT
130 NEXT
140 *TUBE
```

18.9 Tube OSBYTE and OSWORD calls

Read/write Tube flag OSBYTE call

Call address &FFF4
Indirected through &20A
A=&EA (234)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag value is returned in X. A value of 0 indicates that no Tube is present; a value of 255 indicates that a Tube is present.

Read/write byte of i/o processor memory OSWORD calls

Call address &FFF1
Indirected through &20C
A=&05 (5) read i/o processor memory
A=&06 (6) write i/o processor memory

Entry parameters:

X and Y registers contain the address of a 5 byte parameter block. This parameter block contains a 4 byte i/o processor address. The byte read is returned in the last byte of the block. If XY+2=&FE and XY+3=&FF then sideways RAM may be read.

18.10 The Z80 second processor

One of the chief reasons for the immense popularity of this particular microprocessor is the adoption of Digital Research's CP/M operating system by many manufacturers using this microprocessor for their microcomputers. Software designed to run under CP/M requires very little modification to allow it to be used on a wide variety of CP/M machines. Acorn have adopted CP/M as the operating system on the Z80 second processor so that a vast library of software is potentially available to users. Normally the only problem with using CP/M

software initially designed for use on another machine is obtaining a copy on a disc in the correct disc format.

18.10.1 Operating system calls

The operating system calls implemented on the i/o processor are available for use from the Z80. Calls may be made by Z80 machine code via a jump table starting at address &FFCE. This area of the memory map is RAM on the Z80 processor and so there is no requirement for additional indirection vectors in page 2 as the calls may be intercepted at this level.

All the OS calls, except OSARGS take parameters in Z80 registers A, L and H according to the A, X and Y registers on the 6502. Any calls using the carry flag use the same flag on the Z80.

Interception of any of these calls may be achieved by changing the address field in the jump table entry to point to the replacement routine.

The following calls are implemented:

address	call
&FFFE	INT vector - used by Z80 OS
&FFFC	Event vector
&FFFA	BRK vector
&FFF7	OSCLI
&FFF4	OSBYTE
&FFF1	OSWORD
&FFEE	OSWRCH
&FFE7	OSNEWL
&FFE3	OSASCI
&FFE0	OSRDCH
&FFDD	OSFILE
&FFDA	OSARGS
&FFD7	OSBGET
&FFD4	OSBPUT
&FFD1	OSGBPB
&FFCE	OSFIND
&FFC8	TERM
&FF82	Fault pointer
&FF80	Escape flag

These calls are documented in appendix J of the CP/M 2.2 manual.

18.10.2 The Z80 OSWORD call

An additional OSWORD call is available from the Z80 second processor. An OSWORD call with A=&FF may be used to read or write blocks of the i/o processor's memory.

On entry the H and L registers point to the following parameter block:

HL +	0	&0D
	1	&01
	2 - 5	I/O processor address
	6 - 9	Z80 processor address
	&A - &B	Number of bytes to be transferred
	&C	Operation type, 0=read, 1=write

18.10.3 I/O processor memory usage

Eleven pages of i/o processor memory are used by the Z80 OS and CP/M. This memory should not be corrupted by user programs.

&2500 - &25FF is reserved for the Z80 OS
&2600 - &2FFF is reserved for CP/M

18.10.4 Acorn CP/M disc format

The format of a CP/M disc is defined in logical terms by the CP/M operating system but the physical organisation of data on the disc surfaces is normally hardware dependent. The Acorn CP/M disc format is:

80 tracks per disc surface
10 sectors per track
256 bytes per sector

Each double sided disc drive is regarded by CP/M as a single logical drive accessing 160 tracks numbered from 0 to 159. In order to optimise disc drive performance the following logical to physical track mapping is performed:

Logical CP/M track	**Physical disc track**
0 to 79	0 to 79 (first surface)
80 to 159	79 to 0 (second surface)

The first 3 tracks are reserved for use by the CP/M system.

The disc directory is stored in 4K starting from the 4th track. Up to 128 directory entries are available for each disc. The remainder of the disc may be used to store up to 388K of data or programs.

To enable the use of a physical disc sector size (256 bytes) greater than that of the CP/M logical record size (128 bytes) a process called deblocking is used. The effective physical disc sector size may be considered to be 256 bytes as all disc operations handle two sectors at a time using an appropriate sector skew. The table below defines the logical record to physical sector relationship.

Logical CP/M record (128 bytes)	**Logical disc sector** (512 bytes)	**Physical disc size** (256 bytes)
0	0	0
1	0	0
2	0	1
3	0	1
4	1	4
5	1	4
6	1	5
7	1	5
8	2	8
9	2	8
10	2	9
11	2	9
12	3	2
13	3	2
14	3	3
15	3	3
16	4	6
17	4	6
18	4	7
19	4	7

18.11 The 32016 second processor

The 32016 microprocessor is one of a new generation of of 32/16 bit processors. Its internal structure operates on 32 bits supporting a powerful instruction set. This processor is designed to suit the implementation of high level languages such as Pascal and Fortran. Acorn provide a UNIX like operating system, called PANOS, together with Pascal, Fortran, C, BBC Basic, Lisp and a 32016 assembler *bundled*

with the hardware. The Acorn Cambridge Co-Processor may have 512 or 1024 Kb of RAM while the Acorn Cambridge Workstation may have up to 4 Mb. An operating system kernel is provided in 32 Kb of EPROM and is referred to as PANDORA.

It is beyond the scope of this book to describe the 32016 hardware or software in any more detail as these are covered quite adequately in the documentation provided with the systems. However the following points may be of some use.

The PANOS operating system uses normal Acorn file and disc formats but some characters which appear to be legal in DFS may cause problems when trying to access files under panos. Non alpha-numeric characters are best avoided. PANOS also makes use of directories in a specific way to enable mimicry of three letter filename extensions when using Acorn filing systems.

The original 32016 second processor design was carried out with the intention of marketing an 8 MHz machine. Due to the non-availability of the 8 MHz chip sets, Acorn launched the 32016 with a 6 MHz chip set. Unfortunately an unforeseen problem arose with the use of the normal ADFS software and the slower 32016's which meant that a special version of ADFS had to be written for use with the slower machines. The changes which had to be made meant that there was no room to include the floppy disc drivers in this version of ADFS (i.e. it can only be used with a winchester hard disc).

The best way to circumvent this problem is to replace the 6 MHz chip set with 8 or 10 MHz chips and to replace the 12 MHz crystal with a 16 or 20 MHz one (the crystal frequency is halved for the main clock frequency). Once this is done normal ADFS software works perfectly well with floppy or hard discs.

19 Clocks, timers and CMOS RAM

All Acorn-BBC series machines maintain at least two internal clocks. The Master 128 has an additional *real time clock* which is powered by a battery and keeps its time even when the computer is switched off.

The two main OS clocks, the system clock and the interval timer, are updated every centisecond. They are both 5 bytes long. The system clock is used by BASIC for its TIME function; the interval timer is used to generate the timer crossing zero event (see chapter 7).

19.1 System clock OSWORD calls

Call address &FFF1
Indirected through &20C
A=&1 - Read system clock
A=&2 - Write system clock
X and Y point to a parameter block

The clock value is written to or read from the five bytes at the address specified by the X and Y registers.

XY +	0	Least significant clock byte
	1	
	2	
	3	
	4	Most significant clock byte

19.2 Interval timer OSWORD calls

Call address &FFF1
Indirected through &20C
A=&3 - Read interval timer
A=&4 - Write interval timer
X and Y point to a parameter block

The timer value is written to or read from the five bytes at the address specified by the X and Y registers.

XY +	0	Least significant timer byte
	1	
	2	
	3	
	4	Most significant timer byte

19.3 Read timer state switch OSBYTE call

Call address &FFF4
Indirected through &20A
A=&F3 (243)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old timer state is returned in X.

For each internal clock the operating system maintains two copies. The two copies are incremented alternately, and while one copy is being incremented the other copy is available for use. This ensures that a clock value is not used half-way through being incremented. This flag is toggled between the value 5 and the value10 by the operating system.These values represent offsets from the location &28D, and point to the 5 byte value of the system clock.

19.4 CMOS Real Time Clock

The real time clock on the Master 128 allows the correct time and date to be provided to application programs. To check the current time, the MOS command *TIME prints out the current settings. The Master Compact has no real time clock, but still supports the *TIME function. The time on the Compact will always be printed as;

```
Fri,31 Dec 1999.23.59.59
```

Two OSWORDs are provided to allow the time to be written and read.

19.4.1 Read CMOS clock OSWORD &0E(14)

Call address &FFF1
Indirected through &20C
A=&0E(14) - Read real time clock

This OSWORD provides three different ways for reading the real time clock. The desired function is selected by the value placed at the XY location in the parameter block.

Read clock time and date string

On entry XY = 0

On exit the parameter block contains the date as a 24-byte character string starting at XY+0. The format of the string is:

```
ddd,nn mmm yyyy.hh:mm:ss
```

ddd	three character abbreviation for day (eg. Mon)
nn	the day number (eg. 25)
mmm	the three character month abbreviation (eg. May)
yyyy	the year (eg. 1988)
hh	the hour in 24hr notation (eg. 22)
mm	number of minutes past the hour (eg. 07)
ss	number of seconds (eg. 59) followed by carriage return (&0D)

Read time in BCD format

On entry XY = 1

On exit the parameter block contains the BCD clock value in 7 bytes.

Location	Contains	Range
XY	year	00-99
XY+1	month	01-12
XY+2	day of month	01-31
XY+3	day of week	01-07 (Sun-Sat)
XY+4	hours	00-23
XY+5	minutes	00-59
XY+6	seconds	00-59

Convert BCD time to text time string

On entry XY = 2
The remainder of the parameter block XY+1 to XY+7 contains the time in BCD format (as described above, but note that it is stored from XY+1 and not XY+0).

On exit the parameter block contains the 24 byte time and date string as for the option with XY=0 on entry.

Please note that this conversion OSWORD option does not work with some early external 6502 second processors. Interrupts are enabled during this call.

19.4.2 Write CMOS clock OSWORD &0F(15)

Call address &FFF1
Indirected through &20C
A=&0F(15) - Write real time clock

This OSWORD provides three methods for updating the real time clock. It allows the *time only, date only* or *time & date* to be altered with any one call.

Write new *time* only

On entry XY = 8

XY+1	hh	hours time (00-23)
XY+3	':'	ASCII code 58
XY+4	mm	minutes time (00-59)
XY+6	':'	ASCII code 58
XY+7	ss	seconds time (00-59)

On exit the parameter block is unchanged.

Write new *date* only

On entry XY = 15

XY+1	ddd	day of the week (eg. Tue)
XY+4	CHR$(44)	(comma)
XY+5	nn	date in month (eg. 26)
XY+7	CHR$(32)	(space)
XY+8	mmm	month (eg. Feb)
XY+11	CHR$(32)	(space)
XY+12	yyyy	year (eg. 1995)

On exit the parameter block is unchanged.

Write new *time & date* together

On entry XY = 24
XY+1 to XY+15 = the date string as above
XY+16 to XY+24 = the time string as above

On exit the parameter block remains unchanged.

19.5 CMOS RAM/EEPROM

The Master 128 and Compact both contain areas of non-volatile RAM in which the data can be changed by the 6502. The contents of this memory is not lost when the machine is switched off. Information, such as the default settings for screen mode, ROM selected etc., is stored in this area. On the Master 128 there are 50 bytes of CMOS battery-backed RAM associated with the 146818 real time clock chip (see section 19.6). The Compact does not have a real time clock, but has extra non-volatile memory in the form of either a 128 or 256 byte EEPROM. This device has a limited lifetime of apporximately 10,000 write cycles to each location, but it is very unlikely that information stored here will be altered that much!

Two new OSBYTEs are provided in the Master series machines to read and write data into the non-volotile RAM. These are required because the RAM does not appear directly in the 6502 system memory map. Instead it has to be accessed through the slow databus on the system VIA. The OSBYTEs are perfectly adequate for most applications which only require to change the data in the accessible part of the CMOS RAM or EEPROM.

19.5.1 Read CMOS RAM/EEPROM OSBYTE

Call address &FFF4
Indirected through &20A
A=&A1(161)

Entry parameters:

X = the byte to be read
0-49 on the Master 128
0-127 or 254 on the Compact
X = 255 (Compact only) determines if the EEPROM is 128 or 256 bytes long.

The following table describes the use of the CMOS RAM/EEPROM.

Address		Function
0		Econet station number
1		File server station number
2		File server network number
3		Printer server station number
4		Printer server station number
5	d0-3	Default filing system ROM number
	d4-7	Default language ROM number
6		ROMs 0-7 inserted=1, unplugged=0, ROM0=d0
7		ROMs 8-15 inserted=1, unplugged=0, ROM8=d0
8		Allocated to EDIT ROM
9		Reserved for telecommunications applications
10	d0-2	Default screen mode (0-7)
	d3	Default screen select 0=main, 1=shadow
	d4	Default interlace setting 0=on, 1=off
	d5-7	Default *TV setting (100=-4 to 011=+3)
11	d0-2	Default FDRIVE settings
	d3-5	Default CAPS settings only one to be set at any time! — Shift lock b3=1; No lock b4=1; Caps lock b5=1
	d6	Load DIR at switch ON 0=no, 1=ADFS load DIR
	d7	Default drive 0=floppy, 1=hard disc
12		Keyboard auto-repeat delay
13		Keyboard auto-repeat rate
14		Printer ignore character (selected by 15 bit d1)
15	d0	Default Tube selection 0=no Tube, 1=Tube
	d1	Ignore character 0=yes, 1=no
	d2-4	Default serial baud rate (000=75 to 111=19200)
	d5-7	*FX5 setting (000=*FX5,0 to 100=*FX5,4)
16	d0	Unused
	d1	Default BEEP loudness 0=quiet, 1=loud
	d2	Tube selection 0=Internal, 1=External
	d3	Default scrolling 0=enabled, 1=protected
	d4	Default BOOT mode 0=no boot, 1=autoboot
	d5-6	Default serial data format
17-19		Reserved for ANFS
20-29		Reserved for new Acorn firmware/filing systems
30-39		Allocated to ROMs 0-9
40-49		Allocated for user applications

On exit,

X = corrupt
Y = contents of specified CMOS RAM/EEPROM location

if X was 255 on entry with the Compact, Y returns
Y = 0 with NO EEPROM present
Y = &7F with 128 byte EEPROM present
Y = &FF with 256 byte EEPROM present

19.5.2 Write CMOS RAM/EEPROM OSBYTE

Call address &FFF4
Indirected through &20A
A=&A2(162)

Entry parameters:

X = address in CMOS RAM/EEPROM
Y = the byte to be written at location in X

On exit,

X is preserved
Y is corrupt

19.6 CMOS RAM/RTC hardware (Master only)

The CMOS battery-backed RAM/RTC chip is fitted on Master 128 computers in place of the model B/B+ speech processor or the Master Compact EEPROM. The operating system provides a suitable series of OS calls to set and read the clock and access the battery backed RAM (see previous sections).

For specialised applications it may prove necessary to access the chip directly. This is particularly true for users who wish to use the alarm interrupt feature supported by the chip. The chip is able to generate IRQs to the 65C12 at pre-programmed periods every second, minute, hour or at any specific time during the day. This feature may prove useful in circumstances where an interrupt must be generated at a specific time of day whenever the machine is powered up. We must emphasise that Acorn do not guarantee to support this feature on future versions of the Master. To implement the IRQ feature, it is first necessary to make the link LK4 on the main Master pcb (see section 17.2.3 for link position). This link is un-made when machines are shipped.

CMOS RAM Memory map	
Address	**Function**
0	Seconds
1	Seconds alarm
2	Minutes
3	Minutes alarm
4	Hours
5	Hours alarm
6	Day of week
7	Date of month
8	Month
9	Year
10	Register A
11	Register B
12	Register C
13	Register D
14-63	50 bytes of RAM

Binary or BCD contents

The CMOS RAM address map is illustrated above. There are 64 locations, 50 bytes of which are user RAM, the remaining 14 bytes being time and control registers. The Master uses part of the RAM for storing system variables, as described in section 19.5. The remaining bytes can be allocated to user applications requiring parameter storage over a power-down period. To access one of these locations from a user program you are recommended to use OSBYTEs &A1(161) and &A2(162).

Address locations 0-9 contain the time, alarm setting, and date. Each parameter can be stored in either a binary or a BCD format as defined by the DM bit in register B. For example, suppose you wish to program the value decimal 21. In binary data mode this is &15 whereas in BCD mode it will be &21. These locations can only be accessed directly by manipulating the hardware (as in the example at the end of this section), or indirectly using the OSWORDs covered in section 19.4.

Address location	Function	Decimal range
0	Seconds	0 - 59
1	Seconds alarm	0 - 59
2	Minutes	0 - 59
3	Minutes alarm	0 - 59
4	Hours (12hrs mode) Hours (24hrs mode)	1 - 12 (am) add &80 (pm) 0 - 23
5	Hours alarm (12hrs mode) Hours alarm (24hrs mode)	1 - 12 (am) add &80 (pm) 0 - 23
6	Day of week (Sunday = 1)	1 - 7
7	Day of the month	1 - 31
8	Month	1 - 12
9	Year	0 - 99

Interrupts

There are three separate sources of interrupts to the CPU. The alarm interrupt can be programmed to generate interrupts from once per second to once per day. The periodic interrupt can range from 122µs to 500ms. The end of update interrupt can be used to inform the processor that an update cycle is complete.

The time registers are updated every second. During updating, which lasts for 1.984ms, the CPU cannot access the RTC. The 'UIP' bit is set to '1' whenever an update is in progress.

Registers

There are four registers accessible to the CPU. These are resident at memory locations &0A - &0D in the CMOS RAM address space.

Register &0A - Read/Write except UIP which is read only

b7	b6	b5	b4	b3	b2	b1	b0
UIP	DV2	DV1	DV0	RS3	RS2	RS1	RS0

UIP - indicates that an update is in progress, or will start within 244µs. When clear, this bit indicates that an update is not in progress and will not start for at least 244µs. All accesses to the time data must be effected within 244µs of reading a '0' from the UIP bit.

DV2, DV1, DV0 - control the divider chain from the crystal oscillator. It should always be set to DV2=0, DV1=1 and DV0=0.

RS3,2,1,0 control the periodic interrupt rate as per the table below:

RS3	RS2	RS1	RS0	Periodic interrupt rate
0	0	0	0	None
0	0	0	1	3.90625ms
0	0	1	0	7.8125ms
0	0	1	1	122.07µs
0	1	0	0	244.141µs
0	1	0	1	488.281µs
0	1	1	0	976.562µs
0	1	1	1	1.953125ms
1	0	0	0	3.90625ms
1	0	0	1	7.8125ms
1	0	1	0	15.625ms
1	0	1	1	31.25ms
1	1	0	0	62.5ms
1	1	0	1	125ms
1	1	1	0	250ms
1	1	1	1	500ms

Register &0B - Read/write

b7	b6	b5	b4	b3	b2	b1	b0
SET	PIE	AIE	UIE	SQWE	DM	24/12	DSE

SET - when '0' the update cycle advances the counts of the clock once every second. When set to '1' any update cycle in progress is aborted and the time and calendar bytes can be modified without the time being updated in the midst of modification. The bit must be reset to '0' for normal timer operation.

PIE - When set, enables periodic interrupts at a rate determined by RS3 to RS0 to be generated.

AIE - the alarm interrupt enable bit when set to '1' allows the alarm flag AF to assert IRQ. An alarm interrupt occurs for each second that the three time bytes equal the three alarm bytes. The alarm time bytes can be set to a "don't care" state by loading &FF. For example, loading &FF into the minutes alarm and seconds alarm bytes will allow one interrupt whenever the hour selected by the hours alarm byte is reached.

UIE - the update ended interrupt enable when set to '1' allows the UF bit to assert an IRQ.

SQWE - not used in the Master.

DM - the data mode bit indicates whether time and calendar updates should use binary or BCD data. DM should be set to '1' for binary data and to '0' for BCD data.

24/12 - selects either 24 hour mode '1' or 12 hour mode '0'.

Register &0C - Read only

b7	b6	b5	b4	b3	b2	b1	b0
IRQF	PF	AF	UF	0	0	0	0

IRQF - the interrupt request flag is set to '1' whenever one of the following condition becomes true:

PF = PIE = "1"
AF = AIE = "1"
UF = UIE = "1"

The IRQ state is cleared whenever the interrupt service routine reads from register &0C.

PF - periodic interrupt flag is set to '1' at the end of each period as defined by the rate selection bits RS3 to 0. It is reset to '0' when register &0C is read.

AF - alarm interrupt flag indicates that the current time has reached the alarm time. It is reset by reading register &0C.

UF - update-ended interrupt flag is set after each update cycle. It is reset by reading from register &0C.

b3 - b0 are read as '0's and cannot be written.

Register &0D - Read only

b7	b6	b5	b4	b3	b2	b1	b0
VRT	0	0	0	0	0	0	0

VRT - the valid RAM and time bit is set by reading from register &0D. It is reset to '0' when the power is turned off.

b6-b0 are always read as '0's.

The following example sets the alarm to interrupt every minute and produce a beep. Remember that LK4 must be made on the main Master pcb before interrupts from the RTC will be accepted. Note that OSWRCH is used to send the CTRL-G or BELL character to generate a beep. Using OS routines inside interrupt routines is not recommened, although it does work in this particular instance. It would be better to

use one of the supported timer funtions for a serious application. For example, the interval timer crossing zero event could be used with the same effects. The cmos access subroutine is useful because it allows all locations within the CMOS RAM to be written and read.

```
10 DIM M% 200
20 FOR opt%=0 TO 3 STEP 3
30   P%=M%
40   oswrch=&FFEE
50   slowbus=&FE4F              :REM System VIA port A output
60   regb=&FE40                 :REM System VIA port B control
70   ddra=&FE43                 :REM System VIA port A direction
80   [
 90  OPT opt%
100  .init   SEI                \Disable interrupts
110          LDA &206
120          STA oldv           \Save old vector
130          LDA &207
140          STA oldv+1
150          LDA #int MOD 256
160          STA &206           \Replace with new vector int
170          LDA #int DIV 256
180          STA &207           \IRQ2V high byte set
190          CLI
200          RTS
210  .int    LDA &FC            \Interrupt processing routine
220          PHA                \Save registers on the stack
230          PHX
240          PHY
250          LDX #&0C           \Set CMOS RTC interrupt flags address
260          SEC                \SET carry flag to read
270          JSR cmosaccess
280          TYA                \Value returned in Y register
290          BIT #&20           \Test for alarm interrupt flag set?
300          BEQ exit           \interrupt not recognised
310          LDA #7             \BELL character to make a BEEP
320          JSR oswrch         \Using OS routines inside interrupt
330  .exit   PLY                \code is not recommened, but often works
340          PLX
350          PLA                \Pull registers from stack
360          STA &FC
370          JMP(oldv)          \Continue down interrupt chain
380  .oldv   EQUW 0             \space to store old interrupt vector
390  .cmosaccess PHP            \This routine will read or write data
400  \to/from CMOS RAM, including registers not directly accessible
410  \through the OSCLI Address to r/w in X, C=1 to read, C=0 to
420  \write. Y is the byte to write, or the byte read back.
430          SEI                \Interrupts off
440          BCC write
450  .read   JSR address
460          LDA #&49           \select read
470          STA regb
480          STZ ddra           \set slow databus to input
490          LDA #&4A           \set DS active
500          STA regb
510          LDY slowbus        \value to be read in Y register
520          BRA finish
530  .write  JSR address
540          LDA #&41           \select WR on addressable latch
550          STA regb
560          LDA #&4A           \take DS active
570          STA regb
580          STY slowbus
590          BRA finish
```

```
600   .address LDA #2          \enter with X=selected address
610            STA regb        \Write CE & DS inactive to system via
620            LDA #&82        \AS active
630            STA regb
640            LDA #&FF        \slow databus as output
650            STA ddra
660            STX slowbus     \write register address
670            LDA #&C2        \Take CE active
680            STA regb
690            LDA #&42        \strobe AS low to latch data
700            STA regb
710            RTS
720   .finish  LDA #&42        \deactivate DS
730            STA regb
740            LDA #&02        \deactivate CE
750            STA regb
760            PLP
770            RTS
780   .setup   LDX #1          \Seconds alarm address
790            LDY #0          \Interruptwhenever seconds = 0
800            CLC\Write
810            JSR cmosaccess
820            LDX #3          \Minutes alarm
830            LDY #&FF        \Don't care about minutes value
840            CLC             \Write
850            JSR cmosaccess
860            LDX #5          \Hours alarm
870            LDY #&FF        \Don't care about hours value
880            CLC             \Write
890            JSR cmosaccess
900            LDX #&0B        \Flags register
910            SEC             \Read
920            JSR cmosaccess
930            TYA
940            ORA #&20        \Alarm interrupt enable bit to '1'
950            AND #&A7        \Ensure other interrupts disaled
960            TAY
970            CLC             \Write
980            JSR cmosaccess
990            RTS             \Alarms enabled so return
1000    ]
1010 NEXT opt%
1020 REM Grab the interrupt vector
1030 CALL init
1040 REM Set the interrupt parameters and enable
1050 CALL setup
1060 END
```

20 ADC system

20.1 ADC operating system calls

The model B, B+ and Master 128 computers are all provided with an integral analogue to digital converter system. This system provides four input channels which can each measure a voltage between approximately 0 and 1.8 volts. This ability to read analogue voltages and convert them into a digital format which can be processed by the 6502 is very useful. The most common application is to use the four channels to read the X and Y axes of two joystick control levers. Other applications include measuring anything which can be represented by an analogue voltage, from the light level to air pressure.

Although the Electron in its basic form does not possess an analogue to digital converter, one can be fitted with the Plus 1 expansion. The Master Compact is altogether different. Instead of a true analogue system, it runs an *analogue simulator*. This analogue simulation is supported by the MOS on the Compact, and allows 'Atari style' switchable joysticks to be used instead of the usual analogue devices. There is a switch for left, right, up and down. The values returned by the OS are consistent with values returned by an analogue joystick, and can even be programmed to produce changing values (rather than a fixed value for particular switch combinations). Unfortunately, the Compact cannot measure voltages without the addition of extra hardware.

20.1.1 Read ADC channel OSBYTE call

Call address &FFF4
Indirected through &20A
A=&80 (128)
X>0

The read-buffer-status functions of this call are described in section 9.7.

On the Electron this call will generate an unknown OSBYTE paged ROM service call when passed a positive value in the X register. If this service call is not claimed then the values in page 2 of memory allocated to storing ADC information are returned.

The ADC functions of this call are implemented on the Plus 1 expansion software for the Electron, otherwise this call is implemented identically on the B, B+, Master 128 and the Electron. The Compact returns a value for compatibility, but this has no real meaning.

X=0 - returns the channel number indicating which channel was last used for an ADC conversion. If zero is returned this indicates that no conversion has been completed; this occurs following OSBYTEs &10 and &11 (described below). The two least significant bits of X indicate the status of the fire buttons. This is equivalent to the BASIC ADVAL(0) function.

X=1 to 4 - returns the ADC conversion from the channel specified by X in the X and Y registers. This is similar to BASIC`s ADVAL(1) to ADVAL(4). On the Compact the same value is normally returned by ADVAL(1), (3), (5), (7) and ADVAL(2), (4), (6), (8). If a sprite pointer ROM is fitted to the Compact, the ROM then supplies the values to ADVAL(7) and ADVAL(8) according to the pointer movements.

On exit:

A is preserved
C is undefined

20.1.2 Select ADC channel OSBYTE call

Call address &FFF4
Indirected through &20A
A=&10 (16)

This routine is passed on to paged ROMs in the Electron as an unknown OSBYTE paged ROM service call and is implemented in the Plus 1 expansion software. On the Compact it is possible to set the number of channels to be sampled to 128, but the only meaningful values are 2 or less.

Entry parameters:

X value selects number of channels sampled

X=0 Sampling disabled
X=n Number of channels to be sampled
If n>4 then n is taken as 4.

On exit:

A is preserved
X contains the old *FX 16 value
Y and C are undefined

20.1.3 Force ADC conversion OSBYTE call

Call address &FFF4
Indirected through &20A
A=&11 (17)

This routine is passed on to paged ROMs in the Electron as an unknown OSBYTE paged ROM service call and is implemented in the Plus 1 expansion software. Again, this call is implemented on the Compact for compatibility, but it has no real meaning.

Entry parameters:

X value specifies ADC channel

X=n Force ADC conversion on channel n
If n>4 then n is taken as 4

On exit:

A is preserved
If X=0 to 4 it is preserved otherwise X=4
Y and C are undefined

20.1.4 Read current ADC channel OSBYTE call

Call address &FFF4
Indirected through &20A
A=&BC (188) Read current ADC channel

This call is implemented on the BBC microcomputer and in the software of the Plus 1 expansion for the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The ADC channel currently being converted is returned in the X register. This call should not be used to attempt to force an ADC conversion.

20.1.5 Read maximum ADC channel number OSBYTE call

Call address &FFF4
Indirected through &20A
A=&BD (189)

This call is implemented on the BBC microcomputer and in the software of the Plus 1 expansion for the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The maximum channel number to be used for ADC conversions in the range 0 to 4 is returned in the X register.

20.1.6 Read/write ADC conversion type OSBYTE call

Call address &FFF4
Indirected through &20A
A=&BE (190)

This call is implemented on the BBC microcomputer and in the software of the Plus 1 expansion for the Electron. The Compact uses this call in a different manner.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

B, B+, Master 128 and Electron with Plus 1

A value is returned in the X register indicating the previous ADC type setting.

&00 - default (12 bit)
&08 - 8 bit conversion
&0C - 12 bit conversion

Other values have undefined effects. 8 bit conversion creates values in the same range (0 to &FFFF) but with less precision and two to three times as fast.

Master Compact

The setting of the X parameter bits have the following effects:

b7 0 = update ADVAL values from cursor keys and/or digital joystick
1 = do not update ADVAL from keys/joystick, but allow updating to be handled by a sideways ROM.

b6 If set, key values are entered into the keyboard buffer, depending upon the following ADVAL(0) settings. Note that this feature does not emulate the INKEY function as well, so whilst INKEY -106 will detect if the COPY keyboard key is pressed, it will not recognise the FIRE button on the joystick being pressed.

bit	function	ASCII	key
b7	Joystick RIGHT	&89	cursor RIGHT
b6	Joystick UP	&8B	cursor UP
b5	Joystick DOWN	&8A	cursor DOWN
b4	Joystick LEFT	&88	cursor LEFT
b3	Mouse RIGHT	&7F	DELETE
b2	Mouse MID.	&0D	RETURN
b1	Mouse LEFT	&87	COPY
b0	Joystick FIRE	&87	COPY

b5 0 = return varying emulated numbers to ADVAL(1) etc.
1 = return fixed values to ADVAL(1) and ADVAL(2) as:
LEFT = &FFFF to ADVAL(1)
CENTRE (horizontal) = &7FFF to ADVAL(1)
RIGHT = 0 to ADVAL(1)
DOWN = &FFFF to ADVAL(2)
CENTRE (vertical) = &7FFF to ADVAL(2)
UP = 0 to ADVAL(2)

b4 reserved

b3-0 Emulate analogue speed of joystick movement by returning rapidly or slowly changing values related to setting of joystick. 7 = fastest, 0 = slowest.

20.2 ADC Hardware

The analogue to digital converter chip is a 7002 integrating type of ADC converter. It has four input channels which can be selected under software control. By applying voltages between 0 and Vref (about 1.8 volts) to the channel inputs, a 12 bit binary number proportional to the applied voltage will be generated.

20.2.1 Programming the ADC

An analogue to digital conversion is initiated by writing to the data latch and conversion start register situated at Sheila address &FEC0. The bits written into this register have the following effects:

b0-1 Select input channel CH0 - CH3 and start conversion.

b2 flag input to be set to 0.

b3 0 = select 8 bit conversions (~4ms long)
1 = select 12 bit conversions (~10 ms long)

b4-7 not used

There are three registers which can be read. The status register is located at Sheila address &FEC0. This can be tested for conversion completed status.

b0-1 Returns currently selected channel CH0 to CH3

b2 not used

b3 Returns 0 if in 8 bit conversion mode and 1 if in 12 bit conversion mode.

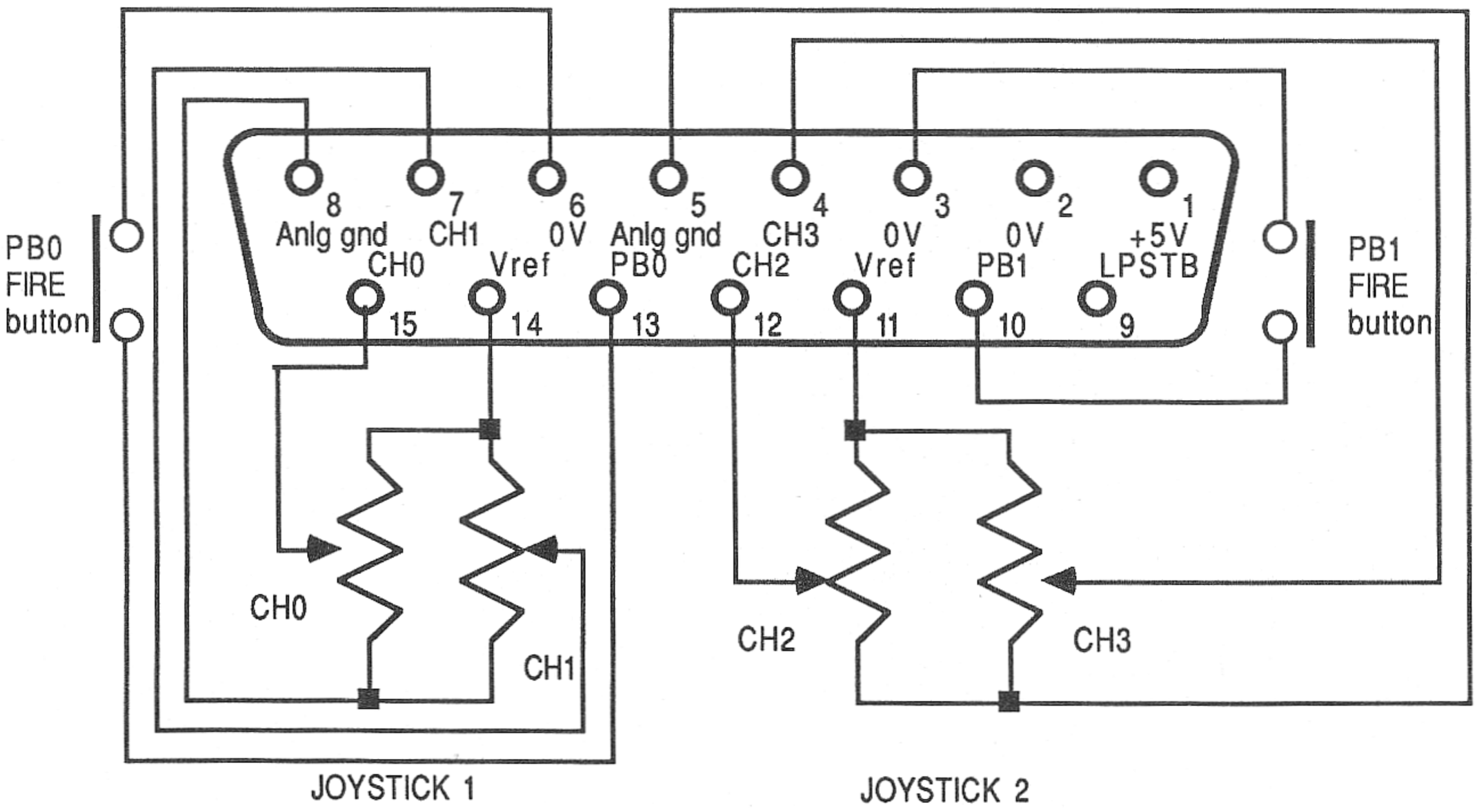
b4-5 Two MSBs of conversion

b6 0 = busy
1 = not busy

b7 0 = conversion completed
1 = conversion not completed

Reading the converted value

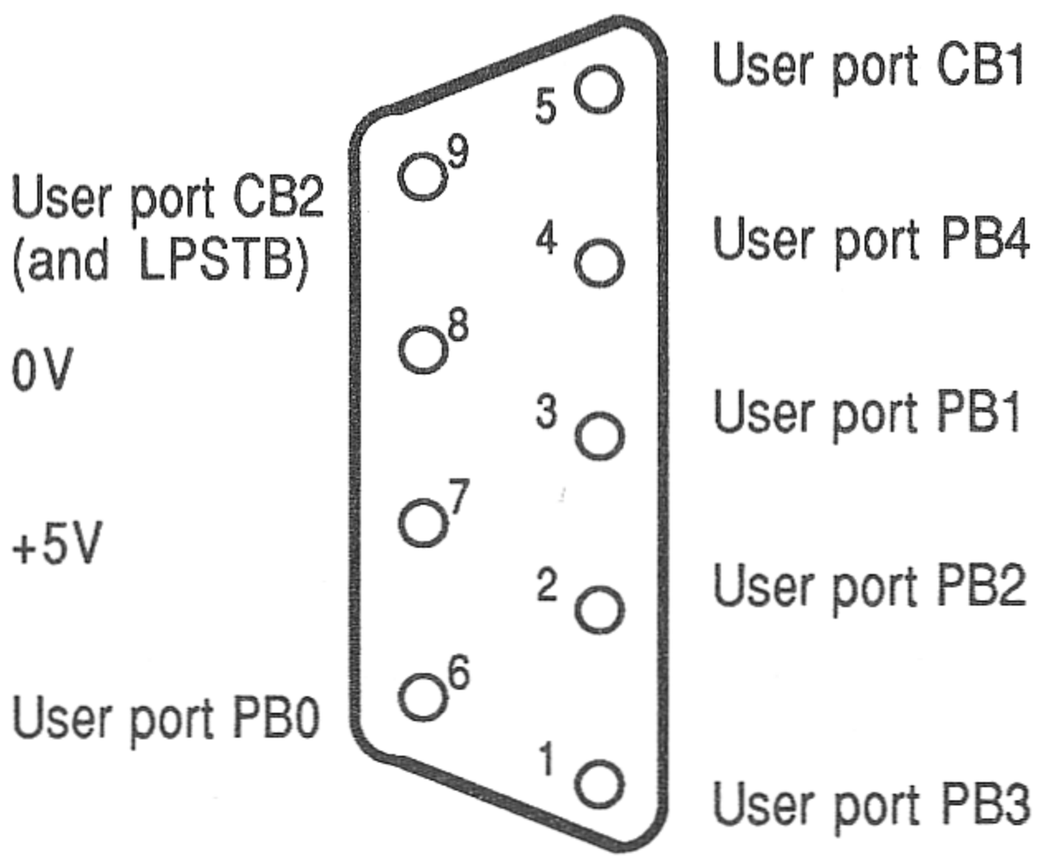
There are two data registers containing the converted value. The high byte is read from Sheila address &FEC1. The four least significant bits are located in bits 7-4 as read from Sheila address &FEC2. These four bits will be highly inaccurate in 8 bit mode, depending upon the qualities of the individual 7002 chips. The values as returned by ADVAL are left to range from 0 to 65535.



View into analogue port showing connections for both joysticks

The analogue port connector on model B, B+ and Master 128 machines is shown above. A pair of joysticks can be connected as shown. Apart from the information needed to construct a joystick, this diagram will be helpful to anyone wishing to be a bit more adventurous with their hardware. One possible use would be the production of a graphics tablet. A series of levers and pulleys connected to a pointer will allow the X-Y position of that pointer to be determined. This allows drawings to be entered into the computer. Some other possible applications include measuring temperature, light level, pH (hydrogen ion concentration), current, voltage, resistance or pressure

Joystick port on the Master Compact



Master Compact Joystick/User port connector

For readers who may want to construct their own Compact joystick devices, the diagram illustrates the allocation of signals to the various pins. For further details about accessing the user port directly, you should refer to chapter 22. The joystick switch allocations are:

PB4 Joystick RIGHT/ mouse Y-axis
PB3 Joystick UP/ mouse X-axis
PB2 Joystick DOWN/ mouse right button
PB1 Joystick LEFT/ mouse middle button
PB0 Joystick FIRE/ mouse left button

21 Sound and speech systems

21.1 Sound and Speech system calls

The following calls have effects on the speech and sound systems.

21.1.1 SOUND command OSWORD call

Call address &FFF1
Indirection address &20C
A=&7
X and Y contain the address of a parameter block.

This routine takes an 8 byte parameter block addressed by the X and Y registers. The 8 bytes of the parameter block may be considered as the four parameters used for the SOUND command in BASIC.

XY +	0	Channel - least significant byte
	1	Channel - most significant byte
	2	Amplitude - least significant byte
	3	Amplitude - most significant byte
	4	Pitch - least significant byte
	5	Pitch - most significant byte
	6	Duration - least significant byte
	7	Duration - most significant byte

This call has exactly the same effect as the SOUND command in BASIC.

21.1.2 ENVELOPE command OSWORD call

Call address &FFF1
Indirection address &20C
A=&8
X and Y contain the address of a parameter block

The ENVELOPE parameter block should contain 14 bytes of data each of which correspond to the 14 parameters described in the ENVELOPE command. This call should be entered with the parameter block address contained in the X and Y registers.

21.1.3 Read/write sound suppression OSBYTE call

Call address &FFF4
Indirected through &20A
A=&D2 (210)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old status value is returned in X.

If this location contains any value other than 0 then sound output is disabled.

21.1.4 Read/write speech suppression OSBYTE call

Call address &FFF4
Indirected through &20A
A=&D1 (209)

This location is not used in the unexpanded Electron and is reserved for future expansion.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old status value is returned in X.

This location contains the value sent to the speech processor when speech is output. A value of &50 represents the SPEAK op. code and is the default value (speech enabled). Writing &20 (NOP) to this location will disable speech.

21.1.5 Test presence of speech processor OSBYTE call

Call address &FFF4
Indirected through &20A
A=&EB (235)

This location is not used in the unexpanded Electron and is reserved for future expansion.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

This location contains 0 if the speech processor is not present and &FF if it is.

21.1.6 Read/write speech processor register OSBYTE calls

Call address &FFF4
Indirected through &20A
A=&9E (158) - read from speech processor
A=&9F (159) - write to speech processor

These calls are implemented on the BBC microcomputer only. On the Electron this call causes the operating system to issue an unknown OSBYTE paged ROM service call but makes no further actions.

Entry parameter:

Data/command in Y for write command

In order to read from the speech ROM a read byte command must have previously been sent to the speech processor using OSBYTE call &9F. If the speech processor has not been primed in this way then a copy of the speech processor's status register is returned in the Y register.

On exit:

A is preserved
X and C are undefined

21.1.7 Read/write CTRL G sound OSBYTE calls

Call address &FFF4
Indirection address &20A
A=&D3 (211) - channel (default value 3)
A=&D4 (212) - amplitude/envelope number (default value 144)
A=&D5 (213) - frequency (default value 101)
A=&D6 (214) - duration (default value 7)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

21.1.8 Electron external sound OSBYTE call

Call address &FFF4
Indirected through &20A
A=&18 (24)

This call is only implemented on the Electron.

This call is used to select an alternative sound system.

Entry parameters:

X contains an undefined parameter

On exit:

A is preserved
All other registers are undefined

21.1.9 Read/write external sound semaphore OSBYTE call

Call address &FFF4
Indirected through &20A
A=&E8 (232)

This call is only implemented on the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old semaphore value is returned in X.

21.1.10 Reset Electron sound system OSBYTE call

Call address &FFF4
Indirected through &20A
A=&74 (116)

This call is only implemented on the Electron. Where it resets the internal sound system. This call has no effect on the B+, Master or Compact. On the BBC model B a bad command error is generated.

21.2 The 76489 sound chip

The sound chip on the BBC microcomputer is in itself a very simple chip. There are three channels for which the frequency and volume of output can be defined. There is also a fourth white noise generator. The output

from all of these channels is automatically mixed on chip. The complex sound commands available from BASIC are very powerful but require a large amount of time to process, especially if complex envelopes are defined. In fast machine code programs it may sometimes be advantageous to write directly to the sound chip. The example program shows how this can be done. The data to be written into the sound chip is first of all put onto the slow databus. Note that interrupts are disabled before this is started. The data defines which sound chip register is updated, and with which parameters. The sound generator write enable line is then pulled low for at least 8 µs then pulled high again to write the data into the sound chip.

Sound chip register address field

R2	R1	R0	Description
0	0	0	Tone 3 frequency
0	0	1	Tone 3 volume
0	1	0	Tone 2 frequency
0	1	1	Tone 2 volume
1	0	0	Tone 1 frequency
1	0	1	Tone 1 volume
1	1	0	Noise control
1	1	1	Noise volume

Tone generators

There are 3 tone generators. The frequency of each channel is determined by 10 bits of data. F9 is the most significant bit. The frequency of each channel can be calculated as :-

$$\text{frequency} = \frac{4{,}000{,}000}{32 \times \text{10 bit binary number}}$$

The volume level for each channel is variable to 16 different levels these are:

Bit				Volume
A3	A2	A1	A0	
0	0	0	0	15 (MAX)
0	0	0	1	14
0	0	1	0	13
0	0	1	1	12
0	1	0	0	11
0	1	0	1	10
0	1	1	0	9
0	1	1	1	8
1	0	0	0	7
1	0	0	1	6
1	0	1	0	5
1	0	1	1	4
1	1	0	0	3
1	1	0	1	2
1	1	1	0	1
1	1	1	1	0 (OFF)

Noise generator

The noise generator comprises a noise source and volume control. The noise generator parameters are defined by three bits.

FB - this bit when set to "0" causes PERIODIC NOISE to be generated. When set to "1" it causes WHITE NOISE to be generated.

Noise frequency control - the noise base frequency can be defined in 4 possible states by bits NF1 and NF0.

NF1	NF0	Frequency
0	0	low
0	1	medium
1	0	high
1	1	tone generator 1 frequency

21.2.1 Programming byte formats

The sound generator is programmed by sending it bytes in the following format:-

Frequency (First byte)

b7	b6	b5	b4	b3	b2	b1	b0
1	R2	R1	R0	F3	F2	F1	F0

Frequency (Second byte)

This byte can be written after the frequency first byte has been written to select the channel.

b7	b6	b5	b4	b3	b2	b1	b0
0	X	F9	F8	F7	F6	F5	F4

Note that the second low order frequency byte may be continually updated without rewriting the first byte.

Noise source byte

b7	b6	b5	b4	b3	b2	b1	b0
1	R2	R1	R0	X	FB	NF1	NF0

Update volume level

b7	b6	b5	b4	b3	b2	b1	b0
1	R2	R1	R0	A3	A2	A1	A0

21.3 Sound chip example program

```
 10 REM Demonstration of direct poke to sound chip
 20 PROCINIT
 30 REPEAT
 40   INPUT"Byte to send to sound chip";A$
 50   A% = EVAL(A$)
 60   CALL DIRECT
 70   UNTIL FALSE
 80 DEF PROCINIT
 90 DIM Q% 40
100 OSBYTE = &FFF4
110 FOR C=0 TO 3 STEP 3
120   P% = Q%
130   [OPT C
140   .DIRECT SEI  \Disable interrupts
150   PHA
160   LDA #&97
170   LDX #&43     \Data direction register A
180   LDY #&FF     \Set all 8 bits as output
190   JSR OSBYTE   \Write to SHEILA OSBYTE CALL
200   LDX #&41     \Output register A
210   PLA
220   TAY          \Y holds byte to sound chip
230   LDA #&97     \Write to SHEILA OSBYTE CALL
240   JSR OSBYTE   \Output to slow data bus
250   LDX #&40     \Output register B
260   LDY #&00     \Set sound chip write pin low
270   JSR OSBYTE
```

```
280    LDY #&08       \Set sound chip write pin high
290    JSR OSBYTE
300    CLI            \Enable interrupts
310    RTS:]
320    NEXT
330 ENDPROC
```

Run the example program and enter &80, &20 and &90 to generate a frequency at maximum volume on channel 3.

21.4 The speech chip (model B and B+ only)

The Speech processor can be added as an optional upgrade. It can be programmed through OSBYTE CALLS &9E, &9F and SOUND &FFxx. The speech data is held in a special serial speech ROM. The standard one provided with the Acorn speech upgrade kit has a selection of words spoken by the news reader Kenneth Kendall. It is also possible to purchase serial ROMs for the speech system which contain games. These plug into the slot on the left hand side of the keyboard. Again, system software is available to read data from these ROMs using OSBYTE calls &9E, &9F and *ROM. For more information about the speech system, refer to the Speech System User Guide.

22 User/printer and system VIAs

There are two 6522 VIAs (Versatile Interface Adapters) inside the Acorn-BBC micros. One of these is dedicated to the MOS and controls the keyboard, sound, joystick fire buttons, CRTC 50Hz, light pen and analogue to digital conversion interrupts, speech (where fitted) and CMOS clock/RAM (Master only). The other drives the parallel printer port and the user port. The 6522 in general will be considered first of all, since it applies to both units. Separate sections on the MOS VIA and the printer/user VIA then follow on with detailed programming and interfacing information.

22.1 6522 Versatile interface adapters in general

Each VIA chip contains two fully programmable bidirectional 8 bit input/output ports. These are designated port A and port B, each one having two handshaking lines for controlling data transfer. There are two 16 bit programmable timer/counters, a serial/parallel or parallel/serial shift register and latched input/output registers.

22.1.1 Pin Descriptions

PA0-PA7 (peripheral port A)

These 8 lines can be individually programmed as inputs or outputs under control of a Data Direction Register. The logic level on the output pins is controlled by an output register and input data can be latched into an internal register under control of the CA1 line. The various modes of operation are all controlled via internal registers which are programmed directly by the 6502 CPU.

CA1, CA2 (port A control lines)

These two lines can act either as interrupt inputs or as handshake outputs. Each line controls an internal interrupt flag which has a corresponding interrupt enable bit. In addition, CA1 controls the latching of data on port A input lines.

PB0-PB7 (peripheral port B)

The 8 bidirectional port B lines are controlled by an output register and a data direction register in a similar way to port A. The logic level of the PB7 output signal can also be controlled by one of the interval timers. The second timer can be programmed to count pulses on the PB6 input.

These outputs are capable of sourcing up to 1 mA at 1.5 volts in the output mode. This allows direct drive of Darlington transistor circuits.

CB1, CB2 (port B control lines)

The port B control lines act as interrupt inputs or as handshake outputs just like port A. They can also be programmed to act as a serial port under the control of the shift register. These lines can only source 100µA.

22.1.2 Electrical specification

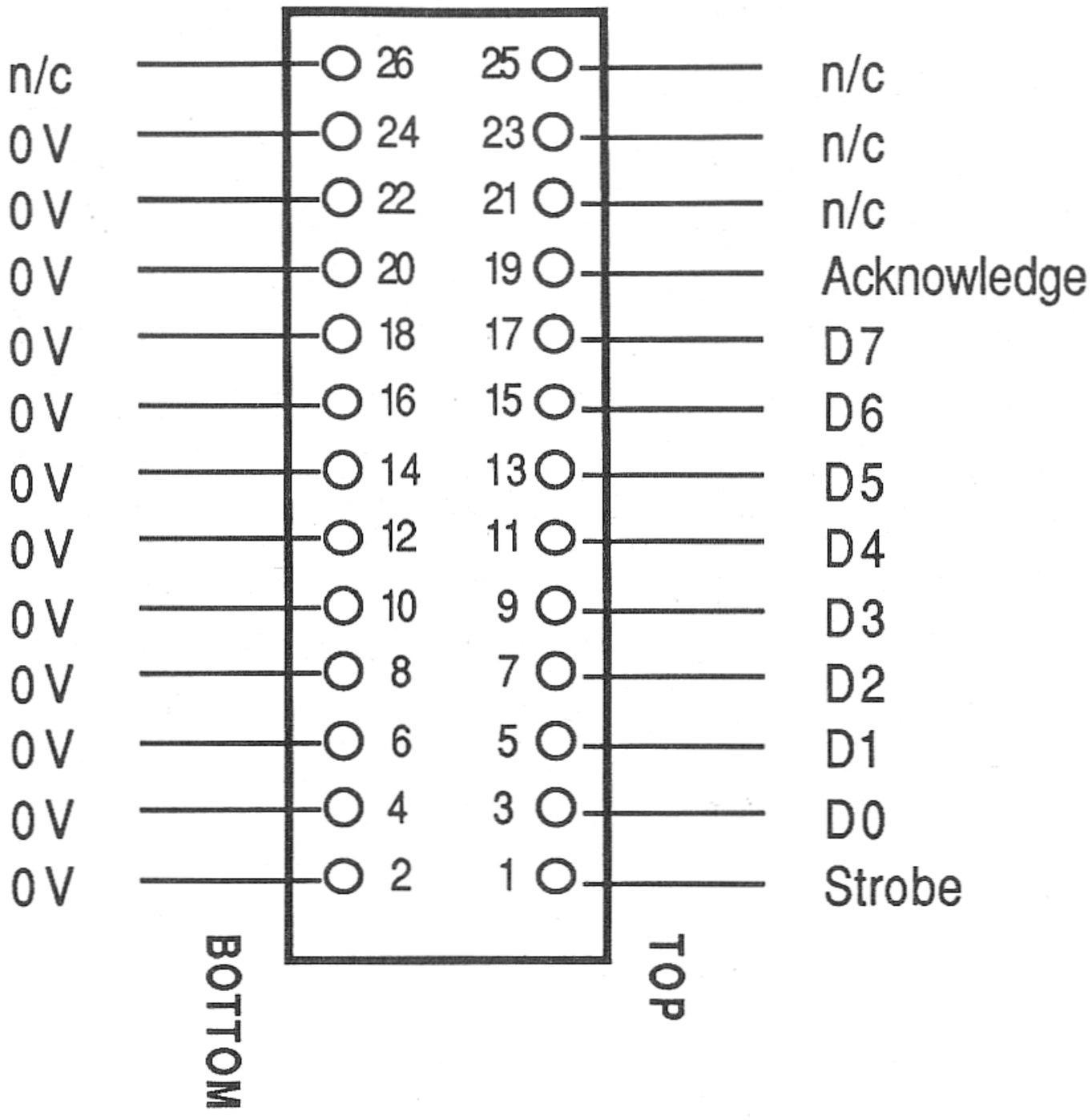
Inputs	
Input voltage for logic 1	2.4 VDC minimum
Input voltage for logic 0	0.4 VDC maximum
Maximum input current	1.8 mA
Outputs	
Output logic 1 voltage	2.4 VDC minimum at a load of 100 µA maximum (except PB0-PB7 1.5 VDC at 1mA)
Output logic 0 voltage	0.4 VDC maximum when sinking up to 1.6 mA

22.2 The User/Printer VIA

Sheila addresses &FE60-&FE6F

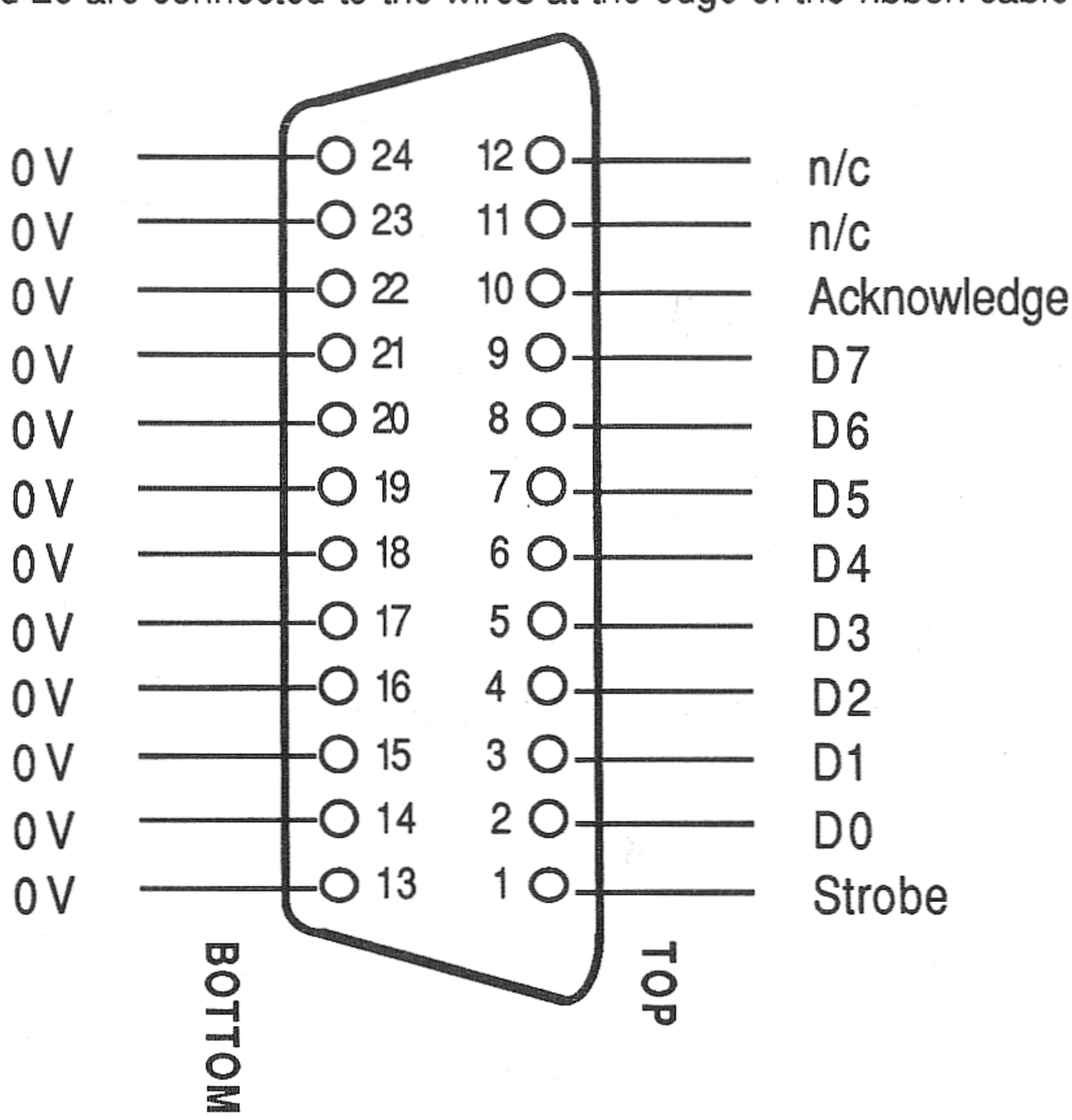
All of the port A lines PA0-PA7 are buffered before being connected to the printer connector. This means that they can only be operated as output lines, but they do have a much larger drive capacity than do unbuffered lines. CA1 can be used directly as described in the general section on 6522s, but note that it is connected to +5 volts via a 4K7 resistor. CA1 normally acts as an "acknowledge" input to the computer from the printer. CA2 usually acts as the printer STROBE output from the computer.

22.2.1 PORT A - The printer port



B, B+ and Master printer port connector looking into socket

Note that pins 1 and 26 are connected to the wires at the edge of the ribbon cable



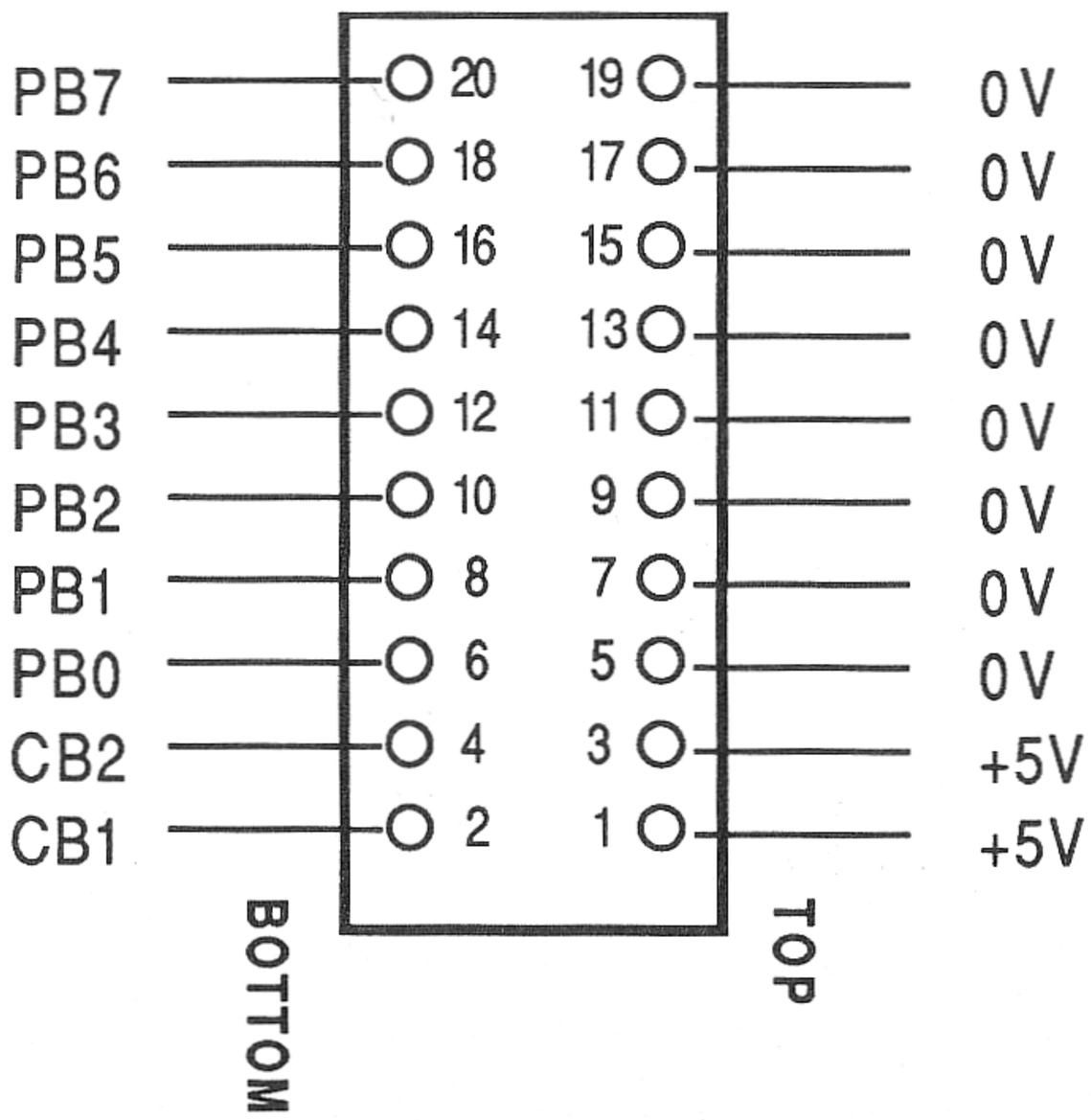
Master Compact printer port connector looking into socket

Note that pins 1 and 24 are connected to the wires at the edge of the ribbon cable

Fig 22.1 - the printer port connectors

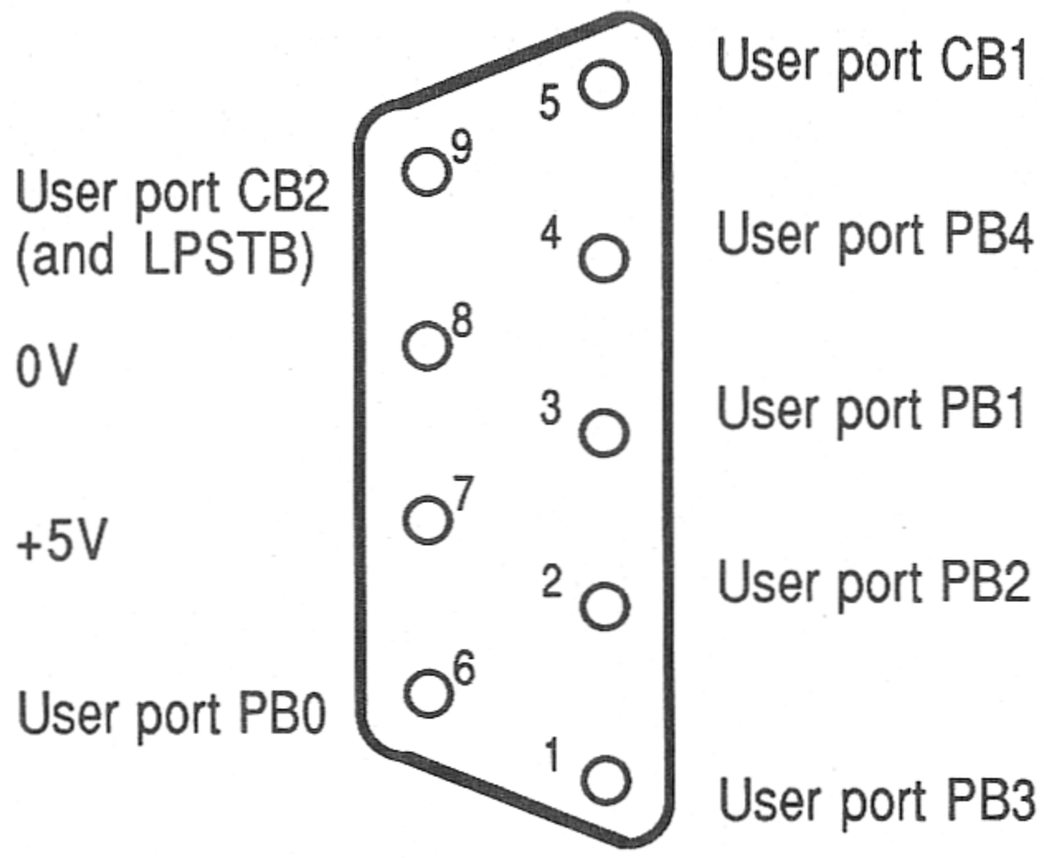
22.2.2 PORT B - The user port

All of port B lines, i.e. PB0-PB7 and CB1, CB2 are available directly on the user port connector. The diagram below (figure 22.2) illustrates the connector. The view is shown looking into the board-mounted connector from outside the case. Wires 1 and 20 are the two outermost wires on the ribbon cable. The "female" part to the connector is a standard 20 way IDC connector. IDC stands for "insulation displacement connector". The plug is normally connected to users' circuits via a length of special 20 way ribbon cable which is available from most good computing or electronics shops. This cable can be connected to a circuit directly by soldering the wires to the circuit board, or indirectly by another IDC plug and header or a DIL header. A DIL header will plug into any ordinary integrated circuit socket.



USER PORT connector looking into socket

Note that pins 1 and 20 are connected to the wires at the edge of the ribbon cable



Master Compact Joystick/User port connector

Figure 22.2 - User port connector

22.3 The System VIA

Sheila addresses &FE40-&FE4F

The System VIA controls the speech system, sound system, keyboard and (only on the Master) the CMOS RAM/EEPROM and clock. Several other functions are also controlled from this VIA. These are the hardware scrolling, vertical sync. pulse interrupt, joysticks input, end of conversion input from the ADC and a light pen strobe input.

22.3.1 System VIA line allocation

PA0-PA7 Slow peripheral data bus

The 6502 CPU does not communicate with the speech system, CMOS RAM, clock, sound generator or keyboard directly over its data bus. Instead, it writes to and reads from the 8 bit port A I/O lines. This forms a "slow" peripheral databus over which the CPU can communicate. To write to this databus, the data direction register A at Sheila &FE43 should set all lines as outputs. The 6502 can then write directly into output register A at Sheila &FE41. To read from the slow data bus, DDRA must set all lines as inputs by writing &00 to Sheila address &FE43. A direct read from input register A at Sheila &FE41 can then be made. NOTE that any reading or writing over this slow databus will have to be done from machine code with ALL 6502 interrupts disabled. This is because the interrupt routines themselves will make extensive use of the system VIA and keep changing the register values. Since the devices hooked onto the slow peripheral bus are well supported by

MOS routines, you are strongly recommended to avoid accessing the hardware directly unless it is absolutely necessary.

CA1 input

This is the vertical sync. input from the 6845. CA1 is set up to interrupt the 6502 every 20 ms (50 Hz) as a vertical sync. from the video circuitry is detected. The operating system changes the display flash colours on this interrupt so that they occur during the screen blanking period.

CA2 input

This input comes from the keyboard circuit, and is used to generate an interrupt whenever a key is pressed.

PB0-PB2 outputs

These 3 outputs drive an 8 line addressable latch which addresses the peripherals attached to the slow peripheral bus.

PB3 output

This bit controls the level of the selected line on the addressable latch.

PB4 and PB5 inputs

These are the inputs from the joystick FIRE buttons. They are normally at logic 1 with no button pressed and change to 0 when a button is pressed. OSBYTE &80 can be used to read the status of the joystick fire buttons.

PB6 and PB7 inputs from the speech processor (model B and B Plus)

PB6 is the speech processor "ready" output and PB7 is from the speech processor "interrupt" output. Speech is also mentioned in section 21.4.

PB6 and PB7 outputs to Master CMOS RAM/RTC

PB6 operates the 146818 chip enable when set to "1". PB7 operates the 146818 address strobe line. For an example using these control lines see section 19.6.

CB1 input

The CB1 input is the end of conversion (EOC) signal from the 7002 analogue to digital converter. It can be used to interrupt the 6502 whenever a conversion is complete. See chapter 20 on the analogue port.

CB2 input

This is the light pen strobe signal (LPSTB) from the light pen. It also connects to the 6845 video processor (see section 13.3) CB2 can be programmed to interrupt the processor whenever a light pen strobe occurs. For more details see the light pen example in chapter 13.

22.3.2 The addressable latch

This 8 bit addressable latch is operated from port B lines 0-3 inclusive. PB0-PB2 are set to the required address of the output bit to be set. PB3 is set to the value which should be programmed at that bit. An example illustrating how to use this latch from BASIC is described in conjunction with the sound generator, see section 21.3. The functions of the 8 output bits from this latch are:-

B0 - Write Enable to the sound generator IC.
B1 - READ select on the speech processor (B and B+)
R/nW control on CMOS RAM (Master only)
B2 - WRITE select on the speech processor
DS control on CMOS RAM (Master only)
B3 - Keyboard write enable

B4,5 - these two outputs define the number to be added to the start of screen address in hardware to control hardware scrolling.

Screen types				No.to add	
Mode	Size	Start of screen	Size	B5	B4
0,1,2	20K	&3000	12K	1	1
3	16K	&4000	16K	0	0
4,5	10K	&5800 (or &1800)	22K	1	0
6	8K	&6000 (or &2000)	24K	0	1

B6 - Operates the CAPS lock LED
B7 - Operates the SHIFT lock LED

22.4 6522 VIAs Functional Description

Register number	Address. for System VIA	Address for User VIA	Register name	Description	
				Write	Read
0	&FE40	&FE60	ORB/IRB	Output register B	Input register B
1	&FE41	&FE61	ORA/IRA	Output register A	Input register A
2	&FE42	&FE62	DDRB	Data direction register B	
3	&FE43	&FE63	DDRA	Data direction register A	
4	&FE44	&FE64	T1C-L	T1 Low-order latch	T1 Low-order counter
5	&FE45	&FE65	T1C-H	T1 High-order counter	
6	&FE46	&FE66	T1L-L	T1 Low-order latch	
7	&FE47	&FE67	T1L-H	T1 High-order latch	
8	&FE48	&FE68	T2C-L	T2 Low-order latch	T2 Low-order counter
9	&FE49	&FE69	T2C-H	T2 High order counter	
10	&FE4A	&FE6A	SR	Shift register	
11	&FE4B	&FE6B	ACR	Auxiliary control register	
12	&FE4C	&FE6C	PCR	Peripheral control register	
13	&FE4D	&FE6D	IFR	Interrupt flag register	
14	&FE4E	&FE6E	IER	Interrupt enable register	
15	&FE4F	&FE6F	ORA/IRA	Same as register 1 but with No handshake	

Figure 22.3 - 6522 Internal Register Summary

22.4.1 Operation of port A and port B

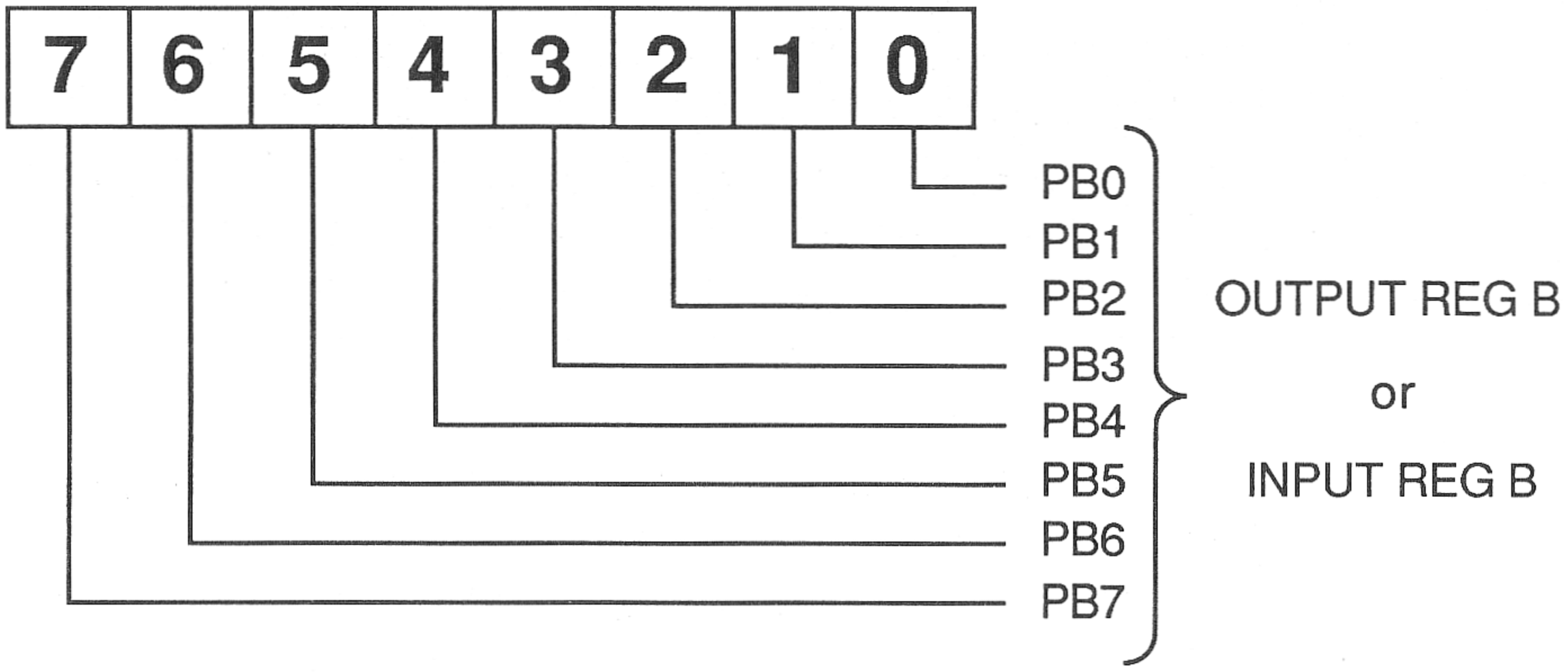
There are two data direction registers DDRA and DDRB which specify whether the peripheral pins are to operate as inputs or outputs. Placing a "0" in a bit of a DDR will cause the corresponding bit of that port to be defined as an input. A "1" will cause it to be defined as an output.

Each of the port's I/O pins is controlled by a bit in an output register (ORA or ORB) and an input register (IRA or IRB). When programmed as an output, a port line will be controlled by the corresponding bit in the output register. If the line is defined as an input then writing data into its output register will have no effect. Reading from a peripheral port will read the value of the input register (IRA or IRB). With input latching disabled IRA will contain the value present at PA0-PA7 when the read is performed. If input latching is enabled then IRA will contain the value present at PA0-PA7 when the latching occurred (via CA1).

The IRB register is similar to the IRA register, but there is a difference for pins programmed as outputs. When reading IRA, it is *the voltage level* on PA0-PA7 which determines the level read back. When reading

IRB, it is always the bit in the output register which is read back. This means that with loads which pull an output "1" low or an output "0" high, reading IRA may indicate a different logic level to that written to the output. Reading IRB will however always read back the value programmed no matter what loading is applied to the pin.

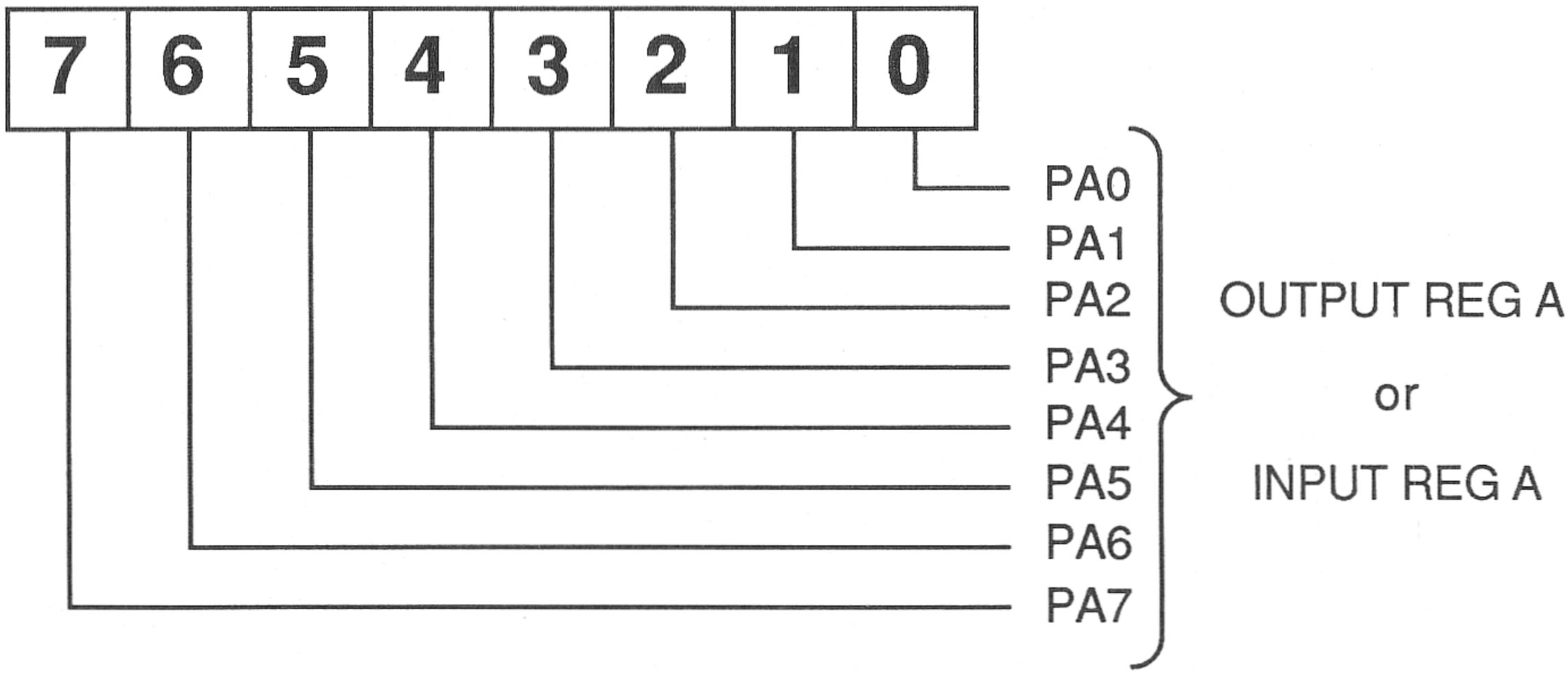
REG 0 - ORB/IRB



PORT B DATA DIRECTION	WRITE FUNCTION	READ FUNCTION
OUTPUT DDRB = "1"	Write output levels to ORB	Reads setting of output register bit
INPUT DDRB = "0"	Writing to ORB does update internal registers but only affects outputs when DDRB is changed	Reads input level on PB pin with input latching disabled.
		Reads IRB bit containing the level at PB pin when CB1 had last active transition if input latching is enabled.

Figure 22.4

REG 1 - ORA/IRA



PORT A DATA DIRECTION	WRITE FUNCTION	READ FUNCTION
OUTPUT DDRA = "1"	Write output levels to ORA	Reads level on PA pin with input latching disabled - not relevant on user VIA as port A is output only to printer.
		With input latching enabled reads the IRA bit reflecting level of PA at last active CA1 transition.
INPUT DDRA = "0"	Writing to ORA does update internal registers but only affects outputs when DDRA is changed	Reads input level on PA pin with input latching disabled.
		Reads IRA bit containing the level at PA pin when CA1 had last active transition if input latching is enabled.

Figure 22.5

REG 2 (DDRB) and REG 3 (DDRA)

Data direction registers port A = DDRA, port B = DDRB

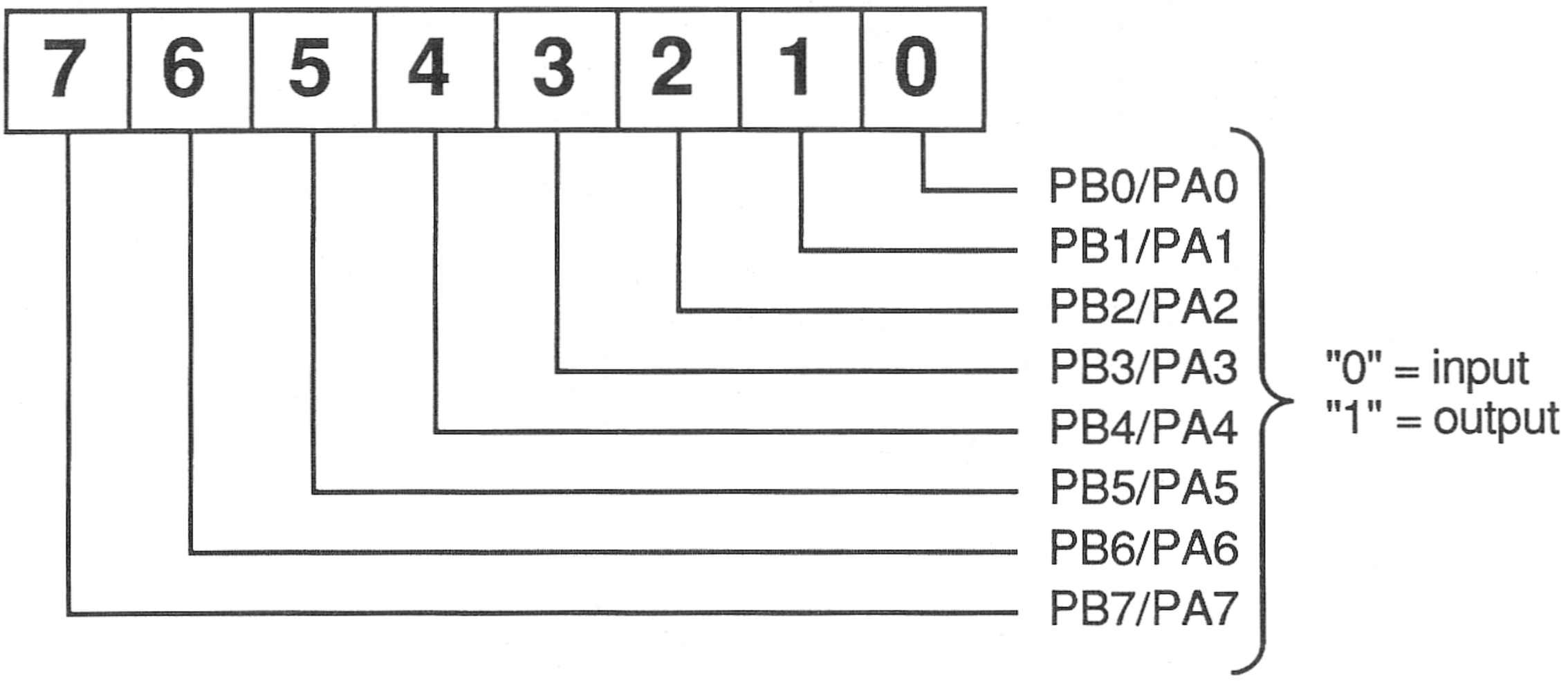


Figure 22.6

22.4.2 Write handshaking data transfer

Handshaking allows data transfers between two asynchronous devices. Write handshaking operates with "data ready" and "data taken" signals. The 6522 provides the "data ready" (CA2 or CB2) signal and accepts the "data taken" (CA1 or CB1) signal from the peripheral device. This "data taken" signal sets the interrupt flag and clears the "data ready" output. See the timing diagram figure 22.7.

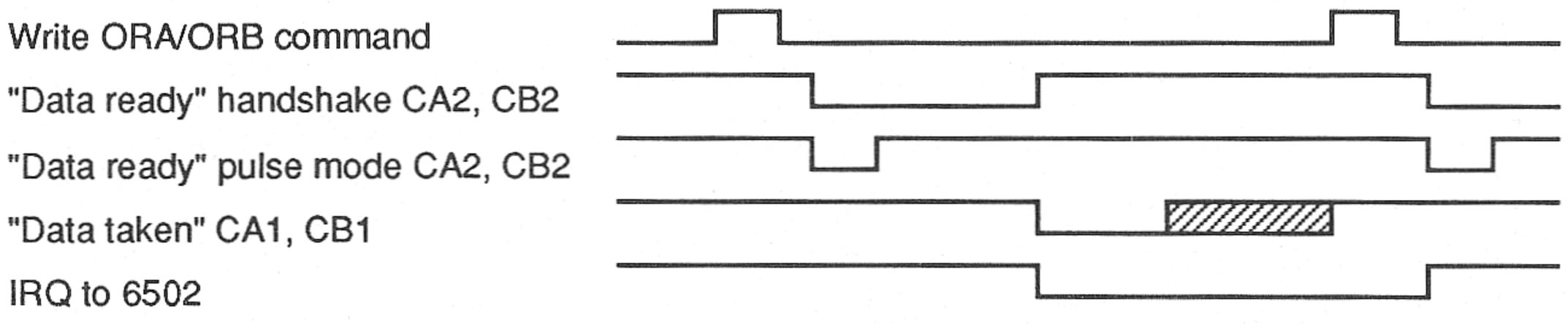


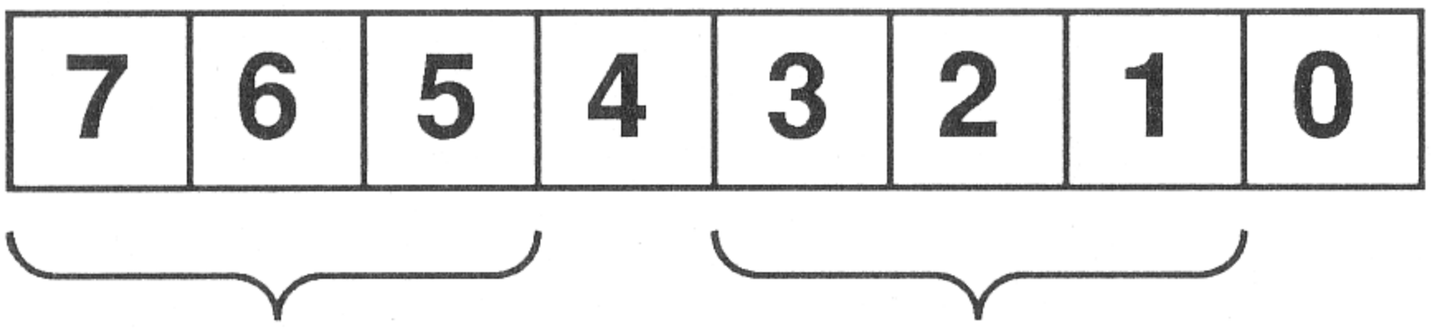
Figure 22.7

Selection of operating modes for CA1, CA2, CB1 and CB2 is controlled by the Peripheral Control Register, see figure 22.8.

REG 12 - The Peripheral Control Register

CB1 Interrupt control	
0	CB1 negative active edge
1	CB1 positive active edge

CA1 Interrupt control	
0	CA1 negative active edge
1	CA1 positive active edge



7	6	5	CB2 Control mode
3	2	1	CA2 Control mode
0	0	0	Negative edges active on input
0	0	1	Independent interrupt, input negative edge
0	1	0	Positive edges active on input
0	1	1	Independent interrupt, input positive edge
1	0	0	Handshake output mode
1	0	1	Pulse output mode
1	1	0	Low output
1	1	1	High output

Figure 22.8

22.4.3 Timer operation

The interval timer, referred to from now on as "T1", consists of two 8 bit latches and a 16 bit counter. After it has been loaded, the counter decrements at the system clock rate (1 MHz) until it reaches zero. When it reaches zero, an interrupt flag will be set and an interrupt will be requested of the 6502, if enabled. The timer then disables any further interrupts, or automatically transfers the contents of the latches into the counter and continues to decrement. The timer may also be programmed to invert the output level on an output line every time its count reaches zero. Figure 22.9 and figure 22.10 illustrate the T1 counter and latches.

REG 4 - Timer 1 low-order counter
REG 6 - Timer 1 low-order latches
REG 8 - Timer 2 low-order counter

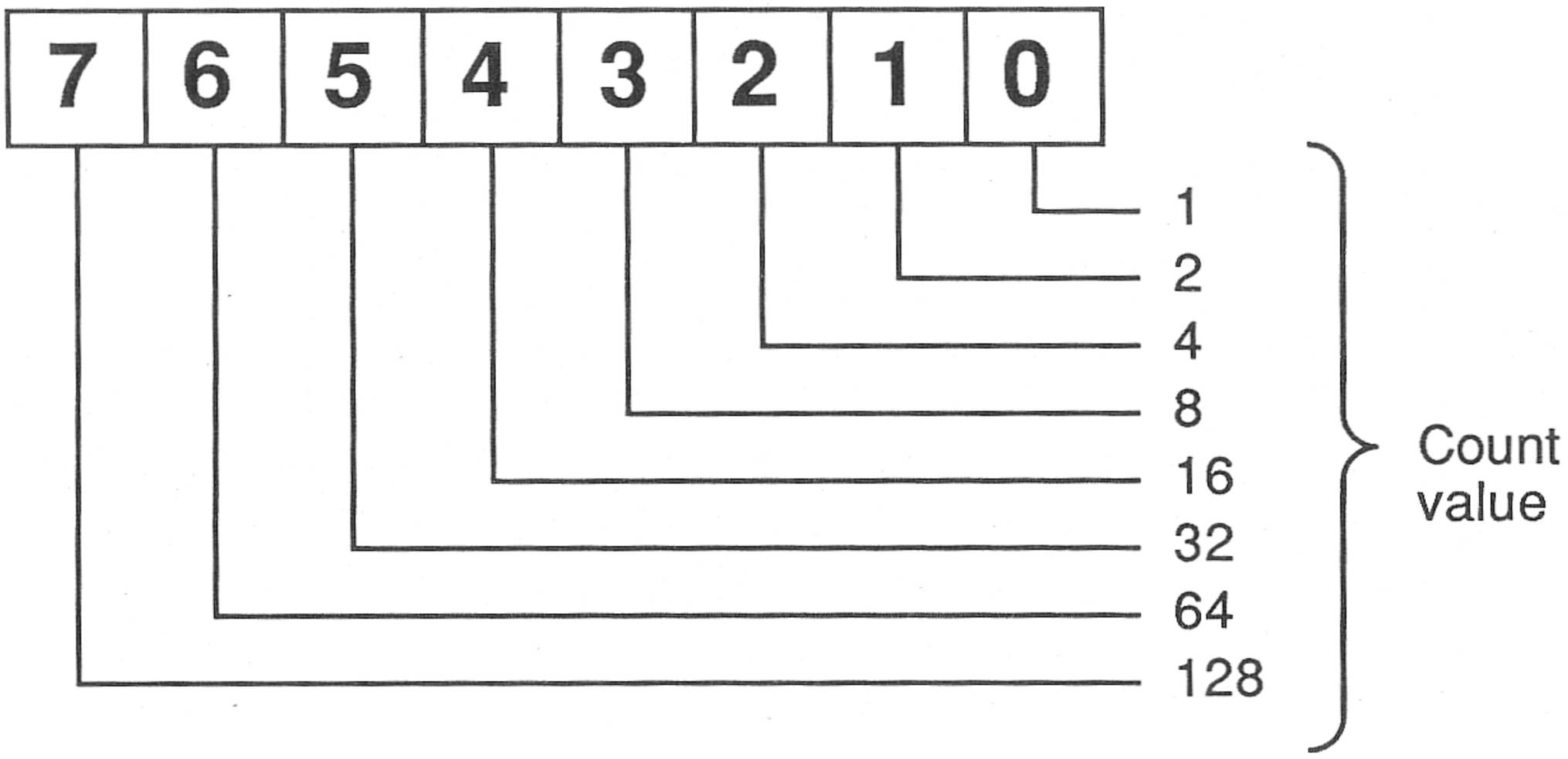


Figure 22.9

REG 5 - Timer 1 high-order counter
REG 7 - Timer 1 high-order latches
REG 9 - Timer 2 high-order counter

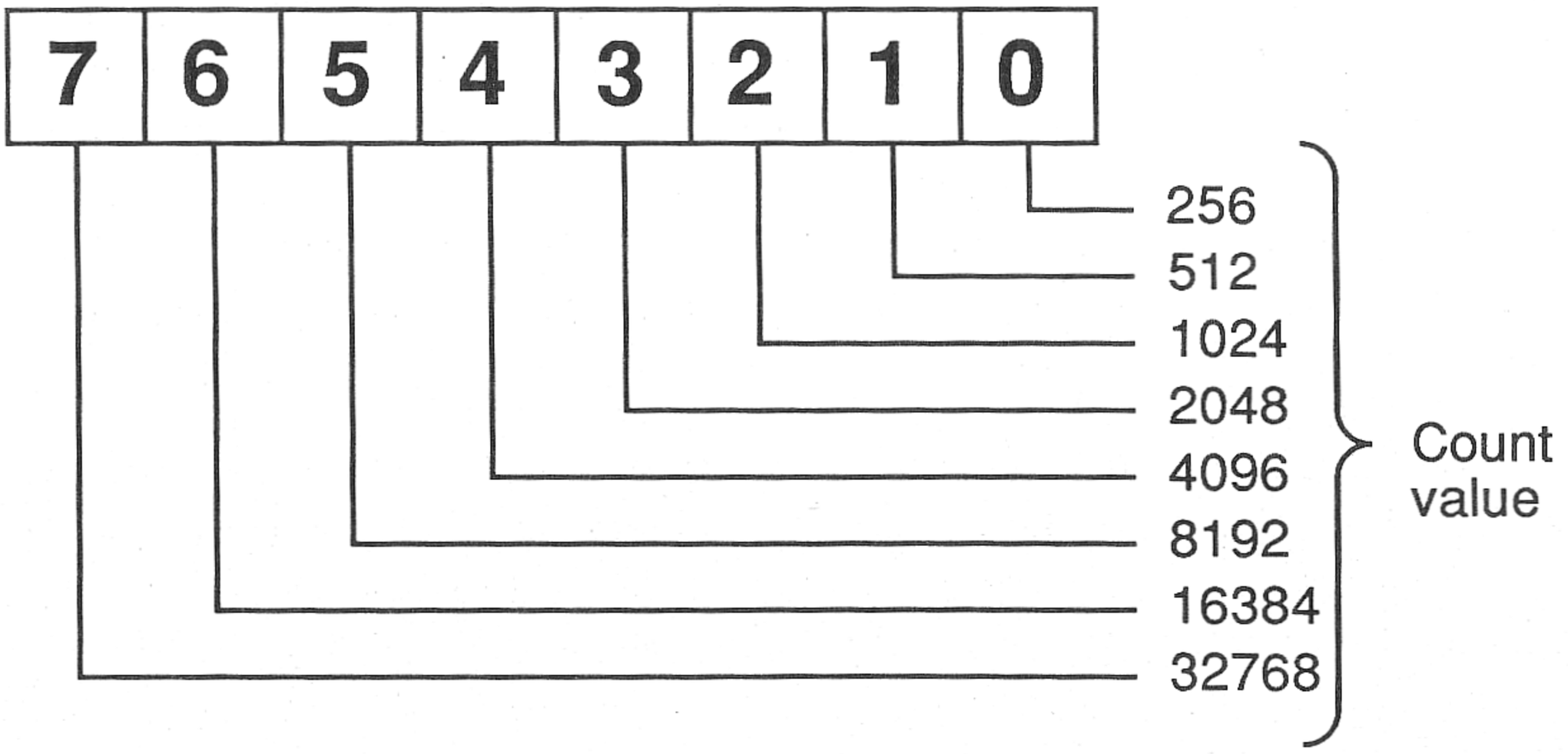


Figure 22.10

22.4.4 Timer 1 one-shot mode

This mode allows a single interrupt to be generated for each timer load operation. The delay between writing T1C-H and generation of the interrupt to the 6502 is a direct function of the data loaded into the counter. T1 can be programmed to produce a single negative pulse on

the PB7 peripheral pin as well as generating a single interrupt. With output enabled (ACR7=1), writing T1C-H will cause PB7 to go low. PB7 will go high again when T1 "times out". The overall result of this is a programmable width pulse on PB7.

Writing into the high order latch has no effect on the operation of T1 in the one-shot mode. It is however necessary to ensure that the low order latch contains the correct data before initiating the count-down by writing T1C-H. When the 6502 writes into the high order counter, the T1 interrupt flag is cleared, the contents of the low order latch are transferred into the low order counter, and the timer begins to decrement at 1MHz. If PB7 output is enabled then it will go low after the write operation. Upon reaching zero, the T1 interrupt flag is set, an interrupt is generated (if enabled) and PB7 goes high. The counter continues to decrement at the system clock rate. The 6502 is then able to read the contents of the counter to determine the time since the interrupt occurred. The T1 interrupt must be cleared before it can be set again.

22.4.5 Timer 1 free-run mode

The advantage of having latches which *remember* the initial value put into the counter is that the initial value can be restored after the counter has decremented to zero. If this is done automatically then the timer enters a free-running mode. In the free-running mode, PB7 is inverted and the interrupt flag is set each time the counter has decremented to zero. The contents of the 16 bit latch are then transferred to the counter, which decrements to zero again and so on. This produces a true square wave of variable frequency on the PB7 output. The interrupt flag can be cleared by writing T1C-H, by reading T1C-L, or by writing directly into the flag.

All of the timers in the 6522 can be retriggered. This means that rewriting the value in the counter will always re-initialise the time-out period. Time-out will therefore be completely inhibited if the processor continues to rewrite the timer before it reaches zero. T1 operates in this way if the 6502 writes into the high order counter (T1C-H). If the 6502 only loads the latches, this will not affect the counter until the next time zero is reached. The timer can be read without affecting its value. This can be very useful because the new timer time doesn't come into effect until zero is reached. If the 6502 responds to each interrupt by programming a new value into the latches, the period of the next half cycle on the PB7 output will be determined. Waveforms with complex mark-space ratios can be generated in this way.

22.4.6 Timer 2 operation

Timer 2 operates either as an interval timer (in the one-shot mode only) or as a counter for counting negative pulses on the PB6 pin. A single control bit in the Auxiliary Control Register selects between these two modes. Timer 2 comprises a "write only" low order latch (T2L-L), a "read only" low order counter and a read/write high order counter. The counter register contents are decremented at 1 MHz. Figure 22.11 illustrates the timer 2 counter registers.

Register	Write function	Read function
REG 4 - T1 low-order counter	Loads T1 low-order latches. Latch contents transferred to low-order counter when high-order counter Reg 5 is loaded.	Reads low-order counter and resets T1 interrupt flag bit in the interrupt flag register.
REG 5 - T1 high-order counter	Loads T1 high-order latches and transfers high and low order latch contents to counter and resets T1 interrupt flag.	Reads T1 high-order counter contents.
REG 6 - T1 low-order latch	Loads T1 low-order latch in identical manner to writing Reg 4.	Reads low-order latch but does not reset T1 interrupt flag.
REG 7 - T1 high-order latch	Loads T1 high-order latch but does not transfer latch to counter.	Reads high-order latch contents.
REG 8 - T2 low-order counter	Writes count value into T2 low-order latches.	Reads T2 low-order counter and resets T2 interrupt flag.
REG 9 - T2 high-order counter	Writes T2 high-order counter value and transfers T2 low-order latch contents to low-order counter. Also resets T2 interrupt flag.	Reads T2 high-order counter.

Figure 22.11

22.4.7 Timer 2 one-shot mode

In the one-shot mode, the operation of timer 2 is similar to that of timer 1. T2 provides a single interrupt for each time out after T2C-H had been set. The counter continues to decrement after time-out, but the interrupt is disabled after the initial time-out so that it will not be set again each time that the timer decrements through zero. T2C-H must be rewritten to re-enable the interrupt flag. The interrupt flag is cleared by reading T2C-L or by writing T2C-H.

22.4.8 Timer 2 pulse counting mode

In this mode, T2 counts a predetermined number of negative going pulses applied to PB6. This can be accomplished by first of all loading a number into T2. Writing into T2C-H will clear the interrupt flag and allow the counter to decrement every time that a pulse is applied to PB6. The interrupt flag is set when T2 counts down past zero. The timer continues to decrement with each pulse applied to PB6. T2C-H must be rewritten to allow the interrupt flag to set on subsequent down counts.

REG 11 - The Auxiliary Control Register

T2 Timer control	
0	Timed interrupt
1	Count down with pulses on PB6

PB latch enable	
0	disable
1	enable

PA latch enable	
0	disable
1	enable

7	6	5	4	3	2	1	0

7	6	T1 Control	PB7
0	0	Timed interrupt each time T1 is loaded.	Disabled
0	1	Continuous interrupts.	
1	0	Timed interrupt each time T1 is loaded.	One-shot output.
1	1	Continuous interrupts.	Square wave out.

4	3	2	Shift register control modes
0	0	0	Shift register disabled
0	0	1	Shift in under control of T2
0	1	0	Shift in using 1MHz clock
0	1	1	Shift in using CB1 as input clock
1	0	0	Shift out free-running at T2 rate
1	0	1	Shift out under control of T2
1	1	0	Shift out using 1MHz clock
1	1	1	Shift out using CB1 as clock

Figure 22.12

22.4.9 Shift register operation

The shift register (SR) enables serial data to be transferred into and out of the CB2 pin under the control of an internal modulo-8 counter. Pulses

from an external source can be applied to CB1 to shift a bit into or out of CB2. Alternatively, with proper mode selection, shift pulses generated internally will appear on the CB1 pin for controlling external devices.

The control bits which select the various shift register operating modes are located in the Auxiliary Control Register. The configuration of the SR data bits and the SR control bits of the ACR are illustrated in figure 22.12 and figure 22.13.

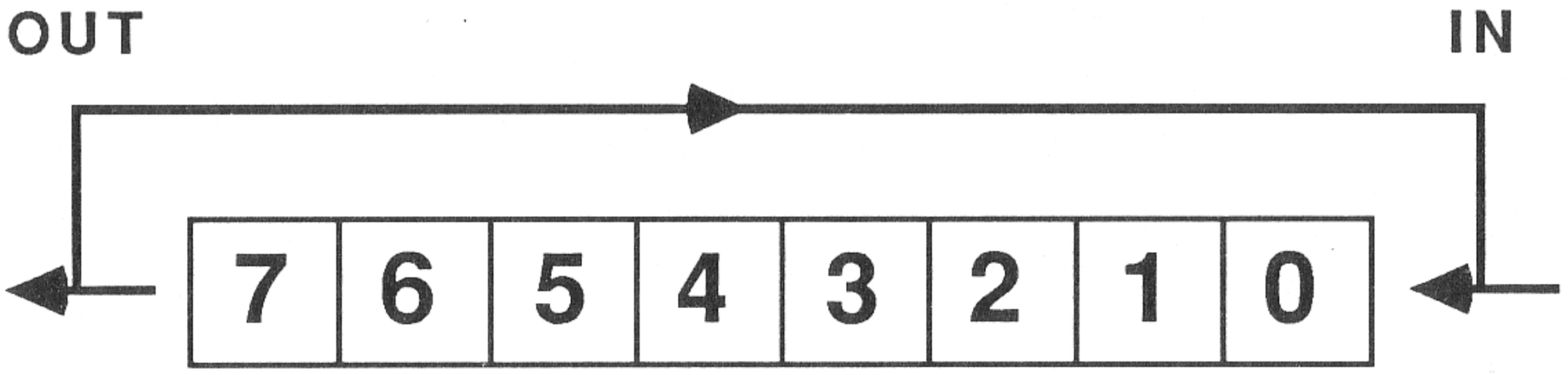


Figure 22.13

22.4.10 Shift register modes of operation

Shift Register Disabled (SRMODE 0)

In this mode the SR is disabled. The 6502 can however write or read the SR and the SR will shift one bit left on each CB1 positive edge. The logic level present on CB2 is shifted into bit 0. The SR interrupt flag is always disabled in this mode.

Shift in under control of T2 (SRMODE 1)

In mode 1 the shifting rate is controlled by the 8 low order bits of T2. Shift pulses are generated on the CB1 pin to control shifting in external devices. The time between transitions of this output clock is controlled by the low order T2 latch.

Reading from or writing to the SR will trigger a shifting operation if the SR flag in the IFR is set. If it isn't set then the first shift will occur when T2 next times out after a read or write SR. Data is shifted first into the low order bit of the SR, then into the next higher order bit and so on on the negative edge of each shift clock pulse. The input data should then change before the next positive going edge of CB1. Data is shifted into the shift register on the positive going edge of the CB1 pulse. After 8

CB1 clock pulses, the shift register interrupt flag will be set and an interrupt will be requested of the 6502.

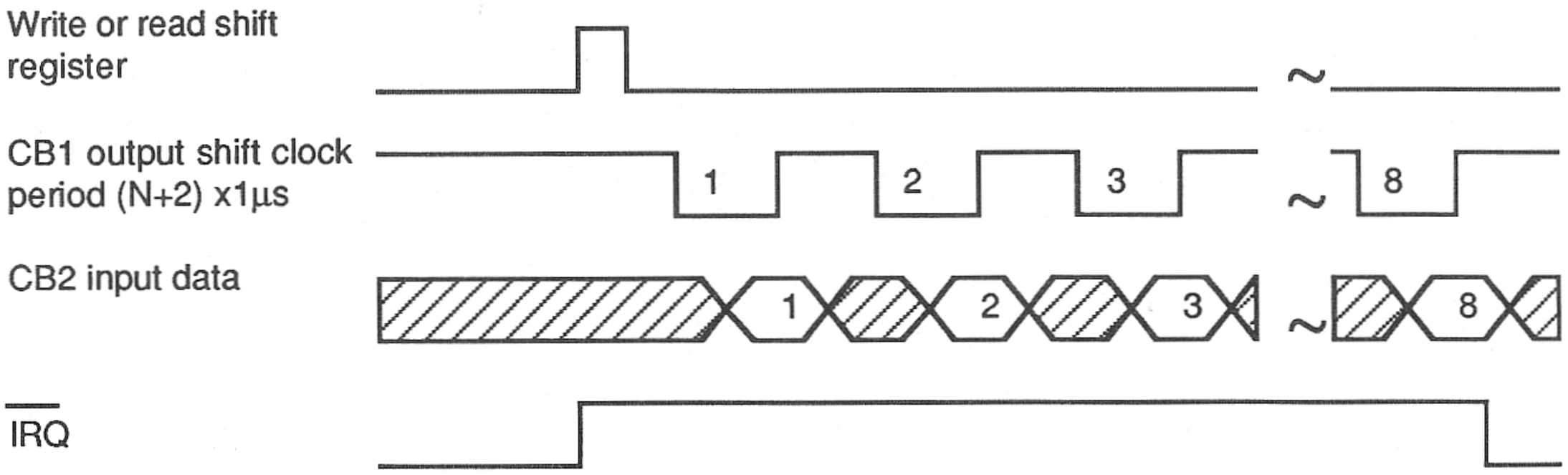


Figure 22.14

Shift in under control of system clock (SRMODE 2)

In mode 2 the shift rate is a direct function of the 1MHz system clock. Pulses for controlling external devices are generated on the CB1 output. Timer 2 has no effect on the SR and acts as an independent interval timer. The shifting operation is triggered by reading or writing the SR. Data is first shifted into bit 0 and then into successively higher order bits on the trailing edges of system clock pulses. After 8 clock pulses, the shift register interrupt flag will be set and output clock pulses from CB1 will cease.

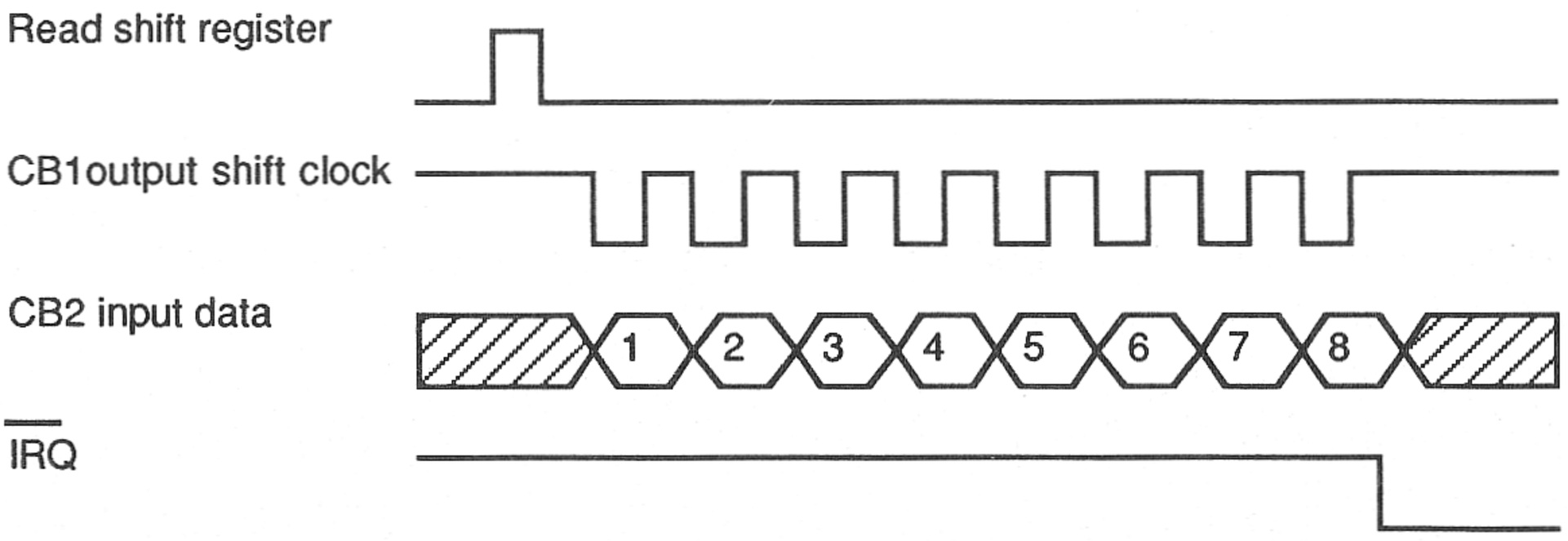


Figure 22.15

Shift in under control of external CB1 clock (SRMODE 3)

CB1 is a clock input in mode 3 so that external devices can load the shift register at their own pace. The shift register counter will generate an interrupt each time that 8 bits have been shifted in. The SR counter does NOT stop the shifting operation, it simply operates as a pulse counter. Reading from or writing to the shift register resets the interrupt flag and initialises the SR counter to count another 8 pulses. Note that data is shifted in on the first system clock cycle following the positive going

edge of the CB1 shift pulse. Data must therefore be held stable during the first full system clock cycle after CB1 has gone high.

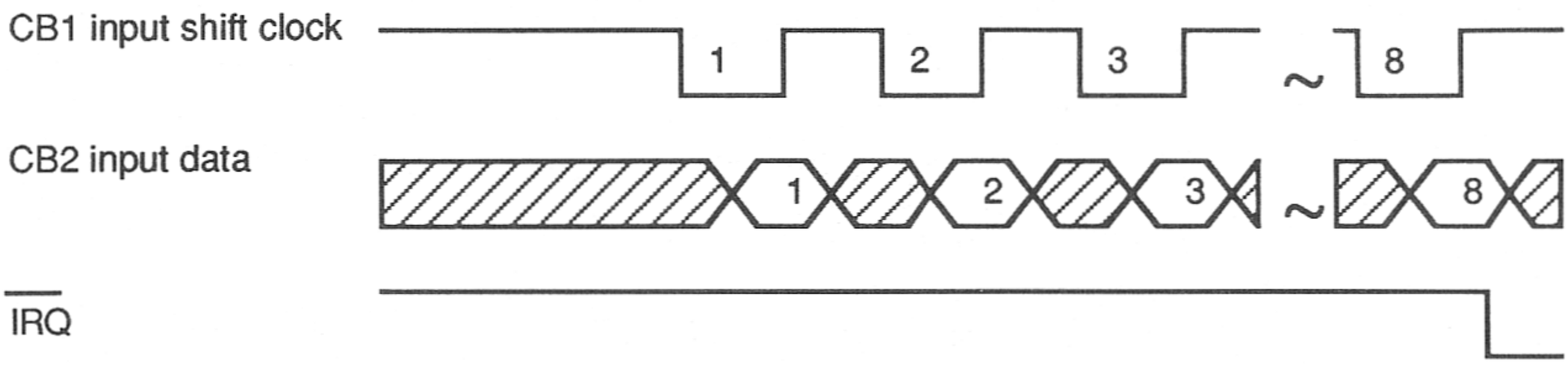


Figure 22.16

Shift out free running at T2 clock rate (SRMODE 4)

In this mode the shift rate is controlled by timer 2 (T2). Unlike mode 5, the SR counter will not stop the shifting operation. Shift register bit 7 is recirculated back into bit 0, so the 8 bits loaded into the shift register will be clocked onto CB2 repetitively. The shift register counter is disabled in this mode.

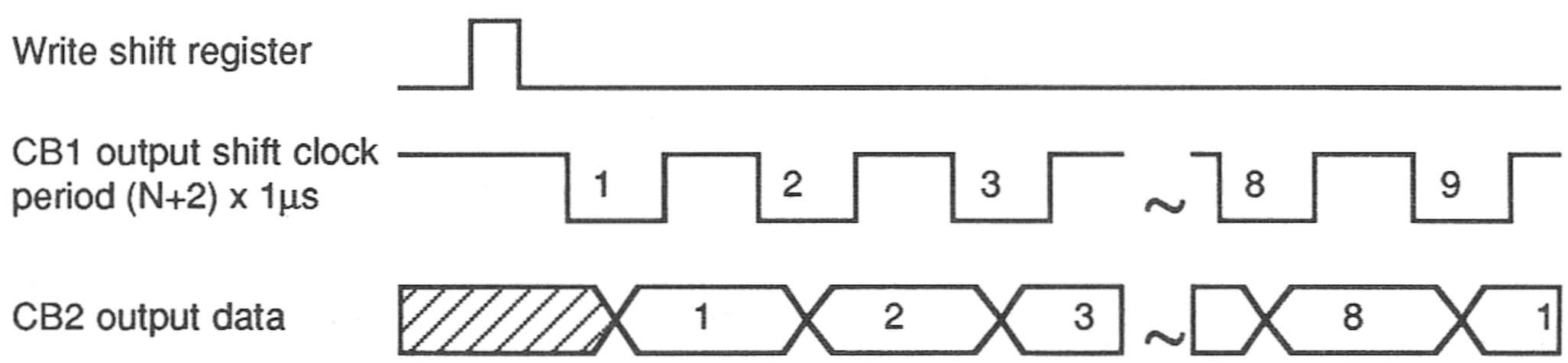


Figure 22.17

Shift out under control of T2 (SRMODE 5)

The shift rate is controlled by T2 as in mode 4. If the SR flag in the IFR is set, then the shifting operation is triggered by the read or write of the SR. Alternatively the first shift will occur at the next timeout of T2 after a read or write of the SR. With each write or read of the SR, the SR counter is reset and 8 bits are shifted onto CB2. Eight shift pulses appear on the CB1 output to facilitate the control of shifting into external devices. When the 8 shift pulses have occurred, shifting is disabled, the SR interrupt flag is set and CB2 remains fixed at the last data bit level.

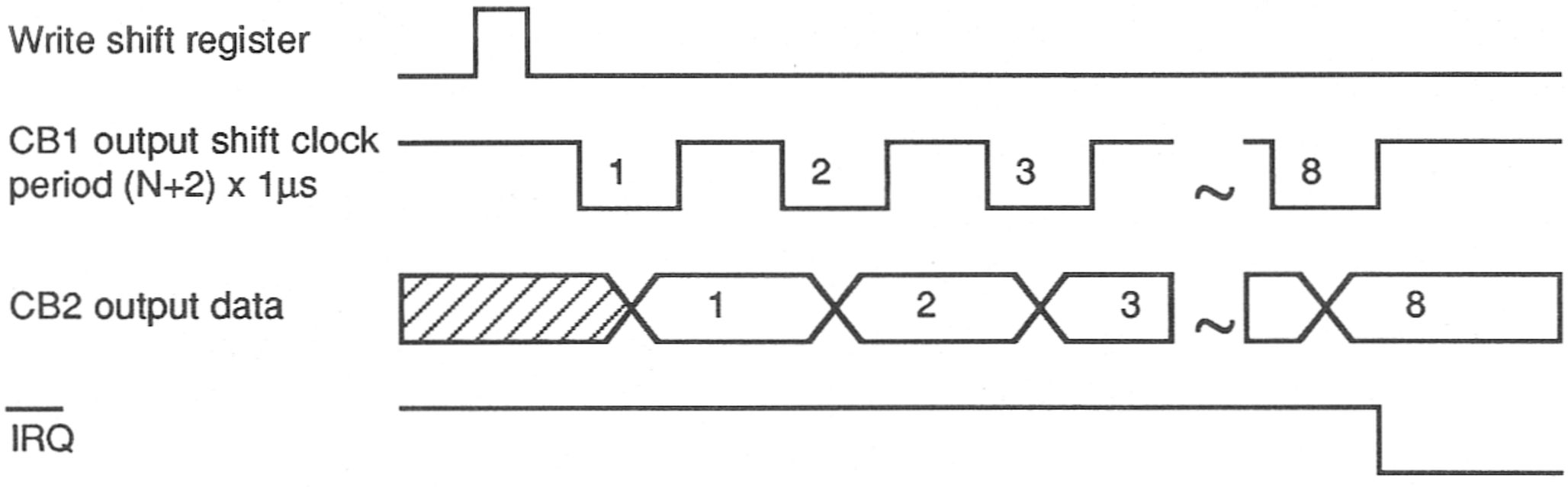


Figure 22.18

Shift out under control of the system clock (SRMODE 6)

In this mode, the shift rate is controlled directly by the 1MHz system clock.

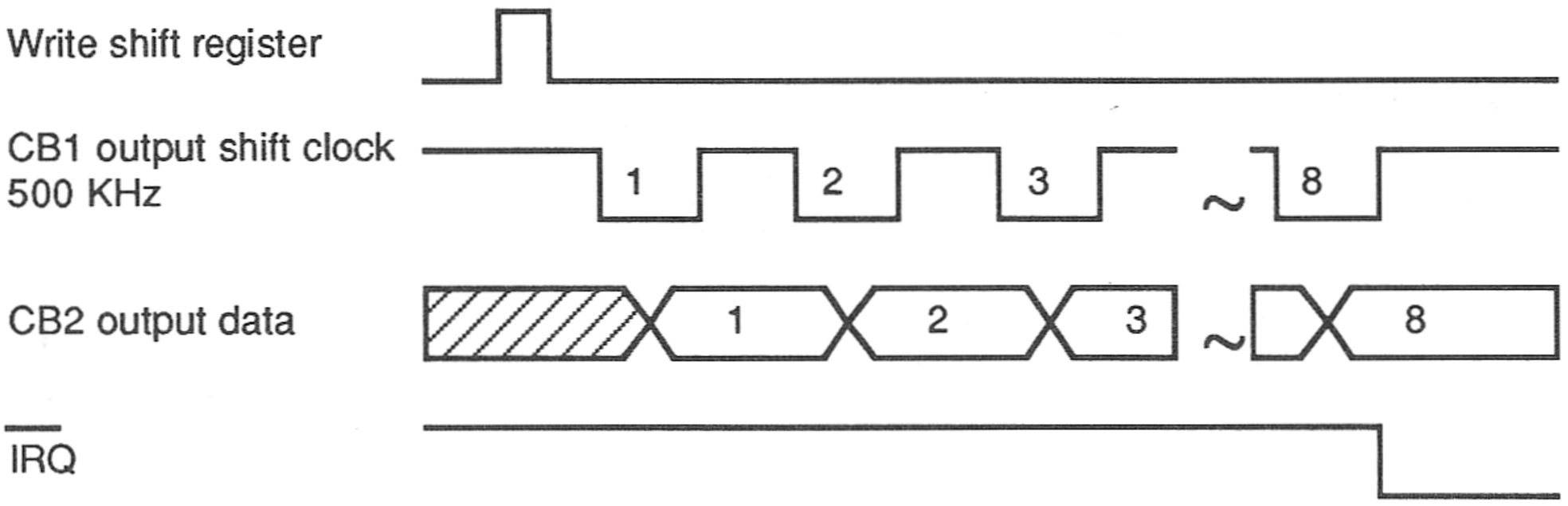


Figure 22.19

Shift out under control of external CB1 clock (SRMODE 7)

In this mode shifting is controlled by pulses applied to the CB1 pin by an external device. The SR interrupt flag is set each time that the SR counter counts 8 pulses, but the shifting function is not disabled. The SR interrupt flag is reset and the SR counter is initialised to begin counting the next 8 shift pulses on CB1, each time that the 6502 writes or reads the shift register. The interrupt flag is set after 8 shift pulses. The 6502 can then load the next byte of data into the shift register.

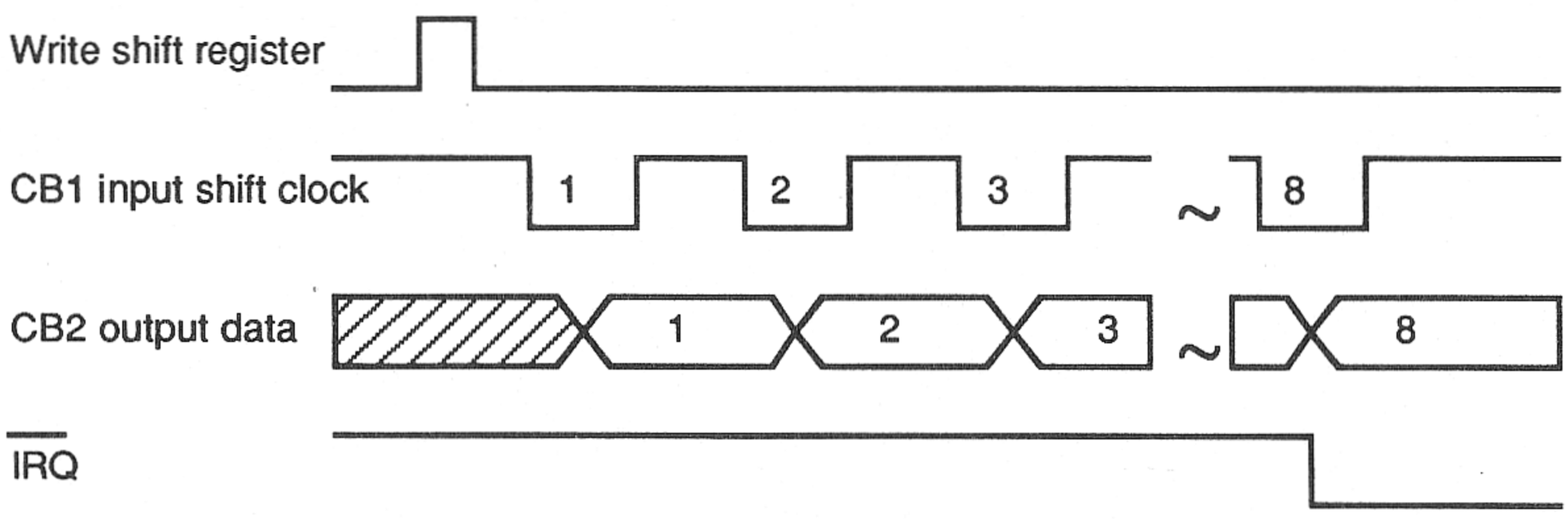


Figure 22.20

22.4.11 Interrupt operation

Interrupt flags are set either by an interrupt condition in the chip (e.g. from a counter), or an interrupt condition on an input to the chip. Interrupt flags normally remain in the set condition until the interrupt has been serviced. The source of an interrupt can be determined by reading these interrupt flags in order from highest priority to lowest priority. This is best performed by reading the flag register into the processor accumulator, shifting either right or left and using conditional branch instructions to detect an active interrupt.

There is an interrupt enable bit associated with each interrupt flag. If this enable bit is set to a logic 1 and the associated interrupt occurs, then the 6502 will be interrupted. If the enable bit is set to 0 then the 6502 will not be interrupted.

All interrupt flags are contained in the interrupt flag register (IFR - see figure 22.21). To enable the 6502 to check the 6522 without checking each bit in the IFR, bit 7 will be set to a logic 1 if the 6522 has generated the interrupt. In addition to reading the IFR, individual bits may be cleared by writing a 1 into the appropriate bit of the IFR. Note however that IFR bit 7 is not a flag as such and will not be cleared by writing a 1 into it. It can only be cleared by clearing all the flags in the register or by disabling ALL of the active interrupts.

The 6502 can set or clear selected bits in the interrupt enable register without affecting the other bits. This is accomplished by writing to the IER. If bit 7 of the byte written is a 0 then each 1 in bits 0-6 will clear the corresponding bit in the IER. For each zero in bits 0-6, the corresponding bit will not be affected. Selected bits can be SET in a similar manner. In this case, bit 7 of the written byte should be set to 1. Each 1 in bits 0-6 will then SET the selected bit. A zero will cause the corresponding bit to remain unaffected. The contents of the IER can be read by the 6502. Bit 7 is then *always* read as a logic 1.

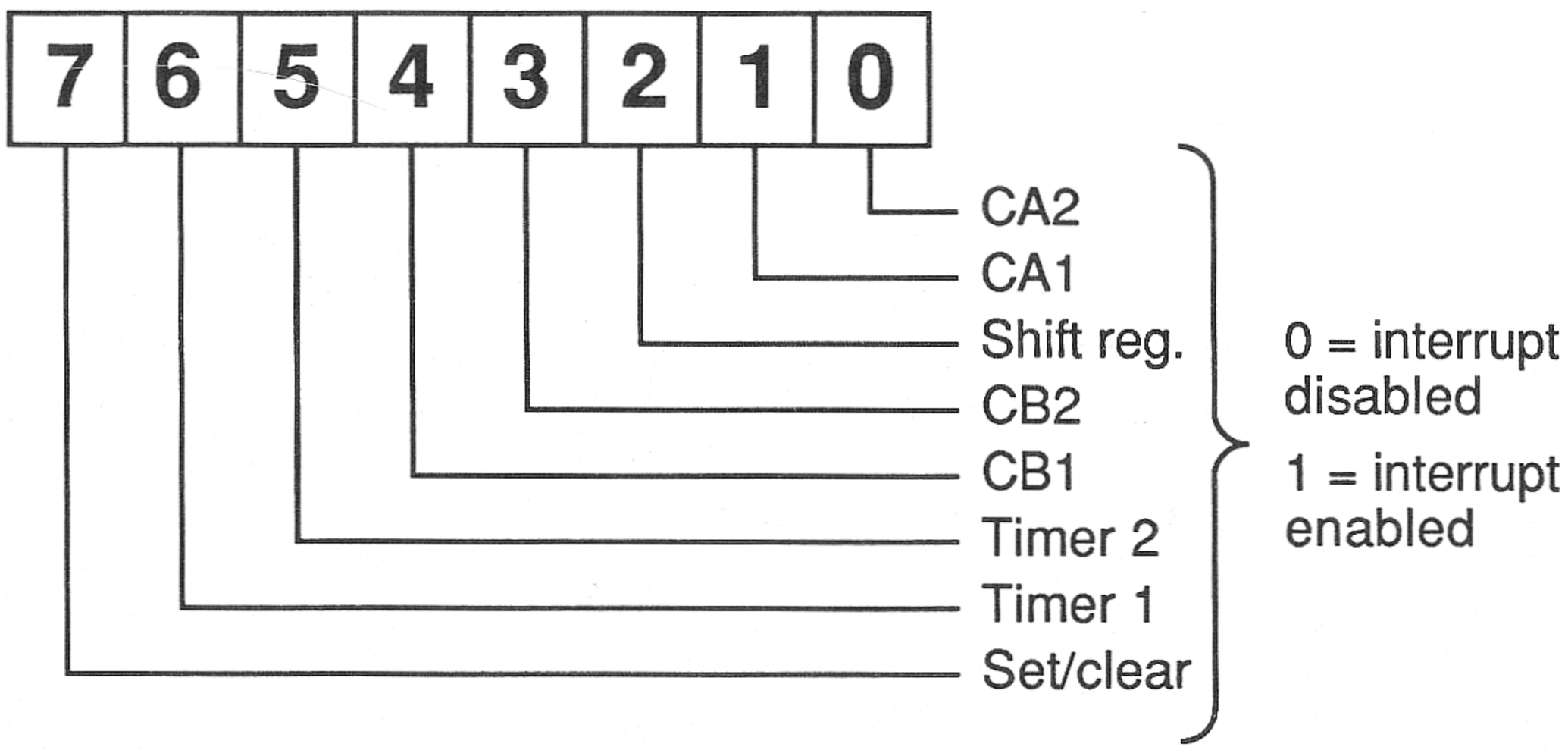
REG 13 - The Interrupt Flag Register

Bit	Set by	Cleared by
0	CA2 Active edge	Read or write reg 1 (ORA)*
1	CA1 Active edge	Read or write reg 1 (ORA)
2	Shift register 8 bits shifted	Read or write shift register
3	CB2 Active edge	Read or write reg 0 (ORB)*
4	CB1 Active edge	Read or write reg 0 (ORB)
5	Time-out of T2	Read T2-low or write T2-high
6	Time-out of T1	Read T1-low or read T1-high
7	Any enabled interrupt	Clear all interrupts

* Reading or writing ORA/ORB will not clear the flag bit if the CA2/CB2 control in the PCR is selected as "independent" interrupt. The bit must be cleared by writing into the IFR.

Figure 22.21

REG 14 - Interrupt enable register



Notes:
(1) With bit7 = "0" a "1" in bits 0-6 disables the corresponding interrupt
(2) With bit7 = "1" a "0" in bits 0-6 enables the corresponding interrupt.
(3) Reading this register bit7= "1" and other bits show enable/disable state..

Figure 22.22

23 The One Megahertz bus & cartridge interfaces

23.1 Introduction to the 1MHz bus

There are basically two routes which a user can take towards adding his own hardware; the USER port and the 1MHz bus (on the Master series this includes the ROM cartridge slot interfaces). The problem with the USER port is that there are only 8 I/O lines and a couple of control lines. For more complex peripherals, direct access to the 6502 address and data buses is required. This interface is provided by the one megahertz bus.

Physically, the one megahertz bus interface has two different forms. On all machines except the Compact and Electron it is found as a 34 pin connector mounted at the front edge of the main circuit board. This connector is accessed from underneath the keyboard. A buffered databus and the lower 8 bits of the address bus are connected to this socket together with a series of useful control signals. An extended set of signals are also available from the cartridge interface on the Master series and Electron, albeit in slightly varying forms (see later). Whilst the designer could use the one megahertz bus in innumerable different configurations, Acorn has defined how the bus should be used to maintain compatibility with other devices.

The standard uses of the one megahertz bus allow up to 64K bytes of paged memory to be used as well as 255 memory mapped devices (plus the paging register). Page &FC (commonly called "FRED") is normally assigned as the memory mapped I/O page and page &FD (commonly called "JIM") is normally assigned as the 64K memory expansion page. Communication between FRED, JIM and programs should be implemented using OSBYTEs &92, &93, &94 and &95.

Access memory mapped IO OSBYTEs

These OSBYTEs read or write bytes to the three pages of memory mapped IO.

OSBYTE calls			
read	**write**	**Memory addressed**	**Name**
&92(146)	&93(147)	&FC00 to &FCFF	FRED
&94(148)	&95(149)	&FD00 to &FDFF	JIM
&96(150)	&97(151)	&FE00 to &FEFF	SHEILA

Entry parameters:

X= offset within page to be read or written
Y= byte to be written (if write)

On exit:

Y= byte read (if read operation)
A is preserved
C is undefined

23.2 "FRED" and Memory Mapped Hardware

In all Acorn-BBC microcomputers page &FC is reserved for peripheral devices with small addressing requirements. These devices are normally accessed via the 1MHz bus interface.

The Master can also access devices at page &FC in the cartridge slot (as well as being able to access paged RAM and ROMs there). When accesses are made to the cartridge slot rather than the older and slower 1MHz bus interface, these accesses are run from the system 2MHz clock. Since only 1MHz or 2MHz accesses can be selected at any time, it is not possible to use the cartridge interface for memory mapped hardware at the same instant as 1MHz accesses are made. The access speed at these addresses is controlled by the 'IFJ' bit in the Master ACCCON register, and is supported by OSBYTE 107. Take care when devices which generate NMIs are being used on the 1MHz bus. It is necessary to ensure that no interrupts can occur whilst the cartridge is enabled for access because any 1MHz bus devices will then be switched out of page &FC.

The current allocations of space in FRED are:-

Address	1MHz Bus	Cartridge interface
&FC00 - &FC0F	test hardware	currently undefined
&FC10 - &FC13	Teletext	currently undefined
&FC14 - &FC17	currently unused	currently undefined
&FC18 - &FC1F	reserved	reserved
&FC20 - &FC27	IEEE 488 Interface	currently undefined
&FC28 - &FC2F	currently undefined	reserved
&FC30 - &FC3F	Cambridge Ring	reserved
&FC40 - &FC47	Winchester Disc	currently undefined
&FC48 - &FC4F	reserved	currently undefined
&FC50 - &FC5F	currently undefined	currently undefined
&FC60 - &FC72	currently undefined	reserved
&FC73 - &FC7F	currently undefined	currently undefined
&FC80 - &FC8F	test hardware	test hardware
&FC90 - &FC9F	currently undefined	reserved
&FCA0 - &FCAF	currently undefined	currently undefined
&FCB0 - &FCCF	currently undefined	reserved
&FCD0 - &FCDF	currently undefined	currently undefined
&FCE0 - &FCEF	currently undefined	reserved
&FCF0 - &FCFE	currently undefined	currently undefined
&FCFF	JIM paging register	JIM paging register

When designing circuits to add on to the one megahertz bus, the "Not page &FC" (NPGFC) signal together with the lower 8 address lines should be decoded to select the add-on circuit. Note that a "clean up" circuit will be required on the NPGFC signal in most applications. This is described in section 23.5. For very keen constructors who require more than the 63 page &FC locations reserved for User Applications, either page &FD can be used for memory mapped peripherals or other FRED locations can be used. Using reserved FRED locations in this way will mean that the hardware add-ons specified for those locations cannot be added in future if user hardware is already using the slot.

23.3 "JIM" and 64K Paged Memory

23.3.1 General description of JIM

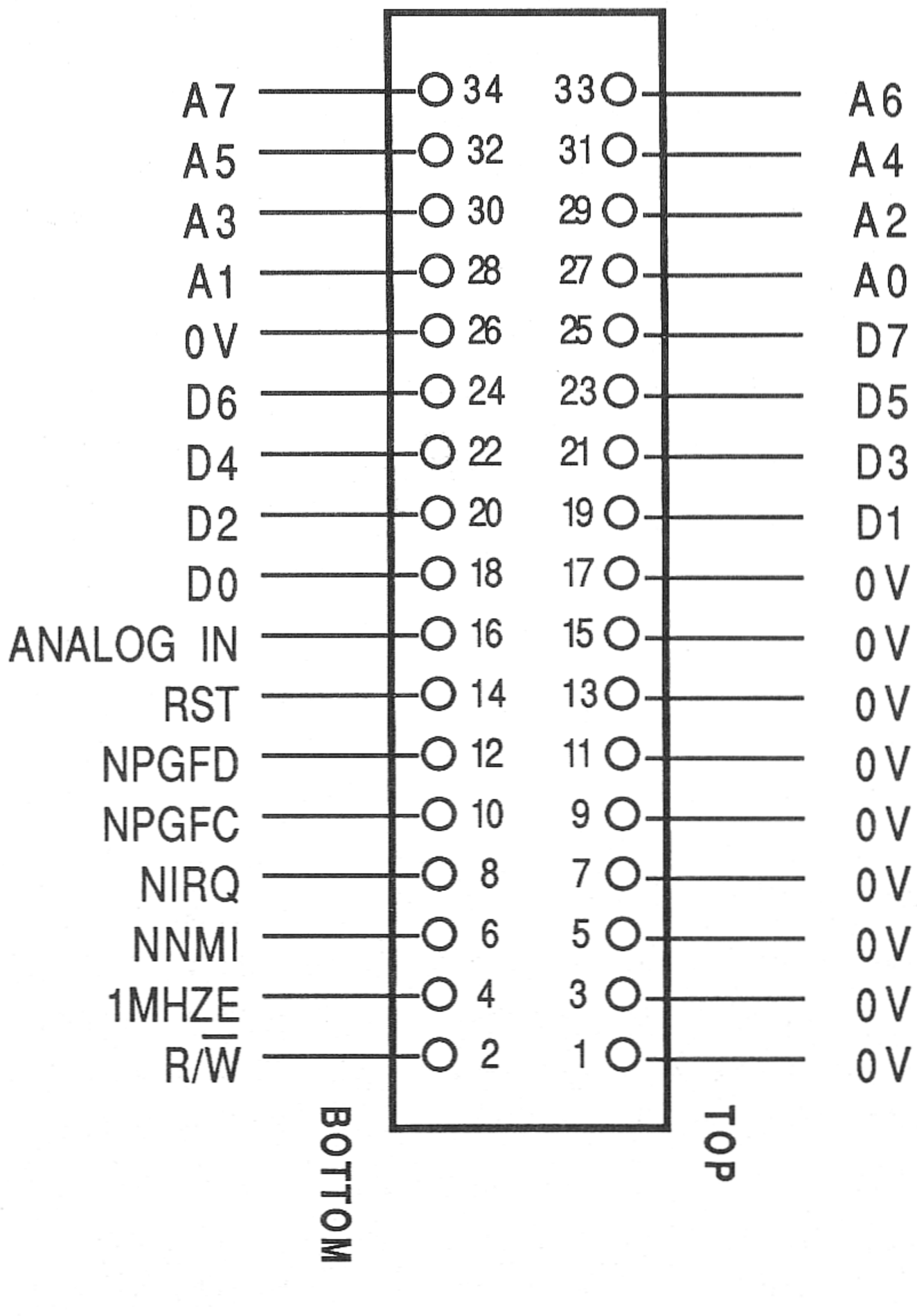
Page &FD in the BBC microcomputer address space can be used in conjunction with the paging register in FRED to provide an extra 64K of memory. This memory is accessed one page at a time. The particular page being accessed is selected by the value in FRED's paging register, and is referred to as the "Extended page number". Note that a "Not

page &FD" (NPGFD) signal is available on the one megahertz bus connector. Accessing memory through the 1MHz bus will generally be much slower than accessing memory directly.

23.3.2 Extended page allocation

"Extended pages" &00 - &7F in JIM are reserved for use by Acorn. The other pages &80 - &FF are reserved for user applications.

23.4 Bus signal definitions



1MHZ BUS connector looking into socket

Note that pins 1 and 34 are connected to the wires at the edge of the ribbon cable

Figure 23.1 - The 1MHz bus connector.

The one megahertz bus connector is illustrated in figure 23.1. The specification for the signals on the one megahertz bus is:-

0 volts This is connected to the main system 0 volts line. The reason for putting 0V lines between the active signal lines is to reduce the interference between different signals.

R/W (pin 2) This is the read-not-write signal from the 6502 CPU, buffered by two 74LS04 inverters.

1MHzE (pin 4) This is the 1MHz system timing clock. It is a 50% duty-cycle square wave. The 6502 CPU is operating at 2MHz, so the main processor clock is stretched whenever 1MHz bus peripherals are being accessed. The trailing edges of the 1MHzE and 2MHz processor clock are then coincidental.

NNMI (pin 6) Not Non-Maskable Interrupt. This is connected directly to the 6502 NMI input. It is pulled up to +5 volts with a 3K3 resistor. Use of Non-Maskable Interrupts on the BBC microcomputer is only advisable after the section on interrupts has been read and thoroughly understood. Both Disc and Econet systems rely heavily upon NMIs for their operation so take care. Note that NMIs are triggered on negative going edges of NMI signals.

NIRQ (pin 8) Not Interrupt Request.This is connected directly to the 6502 IRQ input. Any devices connected to this input should have *open collector* outputs. The line is pulled up to +5 volts with a 3K3 resistor. Interrupts from the 1MHz bus must not occur until the software on the main system is able to cope with them. All interrupts must therefore be disabled after a reset. Note that the main system software may operate very slowly if considerable use is made of interrupts. Certain functions such as the real time clock which is incremented every 10 mS will be affected if interrupts are masked for more than this period. Refer to the chapter on interrupts, chapter 8 for more information.

NPGFC (pin 10) Not page &FC. This signal is derived from the 6502 address bus. It goes low whenever page &FC is written to or read from. FRED is the name given to this page in memory which is described in more detail in section 23.2.

NPGFD (pin 12) Not page &FD. This signal is derived from the 6502 address bus. It goes low whenever page &FD is accessed. JIM is the name given to this page in memory which is described in section 23.3.

NRST (pin 14) Not RESET. This is an active low output from the system reset line. It may be used to initialise peripherals whenever a power up or a BREAK causes a reset.

Analogue Input (pin 16) This is an input to the audio amplifier on the main computer. The amplified signal is produced over the speaker on the keyboard. Its input impedance is 9K Ohms and a 3 volt RMS signal will produce maximum volume on the speaker. Note however that signals as large as this will cause distortion if the sound or speech is used at the same time.

D0 - D7 (pins 18 - 24) This is a bidirectional 8 bit data bus which is connected via a 74LS245 buffer to the CPU. The direction of data transfer is determined by the R/W line signal. The buffer is enabled whenever FRED or JIM are accessed.

A0 - A7 (pins 27 - 34) These are connected directly to the lower 8 CPU address lines via a 74LS244 buffer which is always enabled.

23.5 "Cleaning up" FRED and JIM's page selects

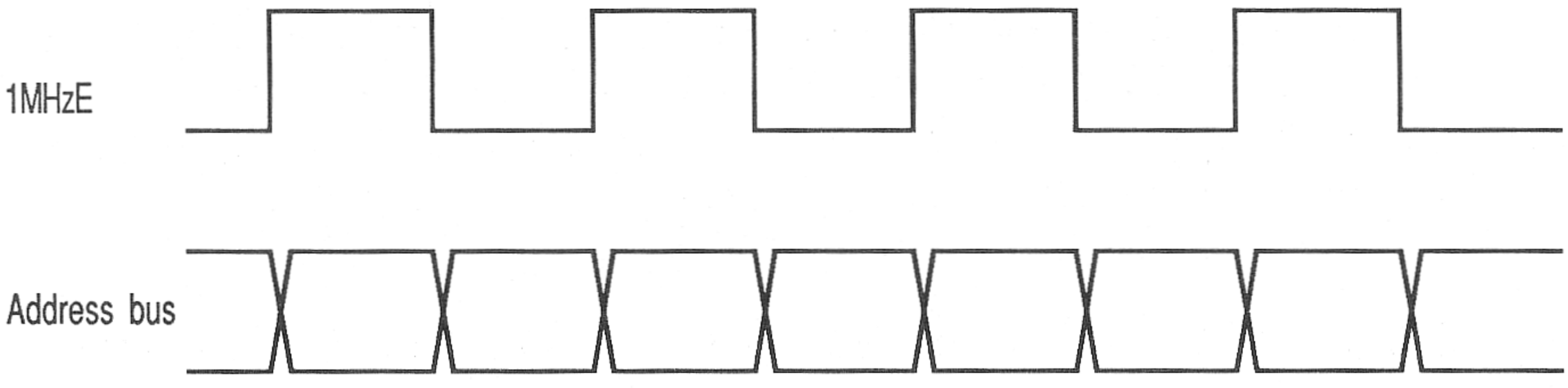


Figure 23.2 - 1MHz bus timing showing page select signals

All 1MHz peripherals are clocked by a 1MHz 50% duty cycle square wave, designated as 1MHzE in figure 23.2. This clock rate was chosen to allow chips such as 6522 VIAs to use their internal timing elements correctly. The system 6502 CPU is normally clocked at twice the speed of the peripherals and so it operates at 2MHz on the model Bs and 4MHz on the Master series (the Master series have a 2MHz peripheral bus as well - see section 23.7). However, if the CPU wishes to access any device on the 1MHz bus, the processor has to be slowed down. The effect of this slow down circuit is illustrated in figure 23.2. After generating a valid 1MHz address, the slow down circuit stretches the clock high period. Unfortunately, two major problems arise from this mode of operation:

23.5.1 Spurious address decoding "glitches" - PROBLEM 1

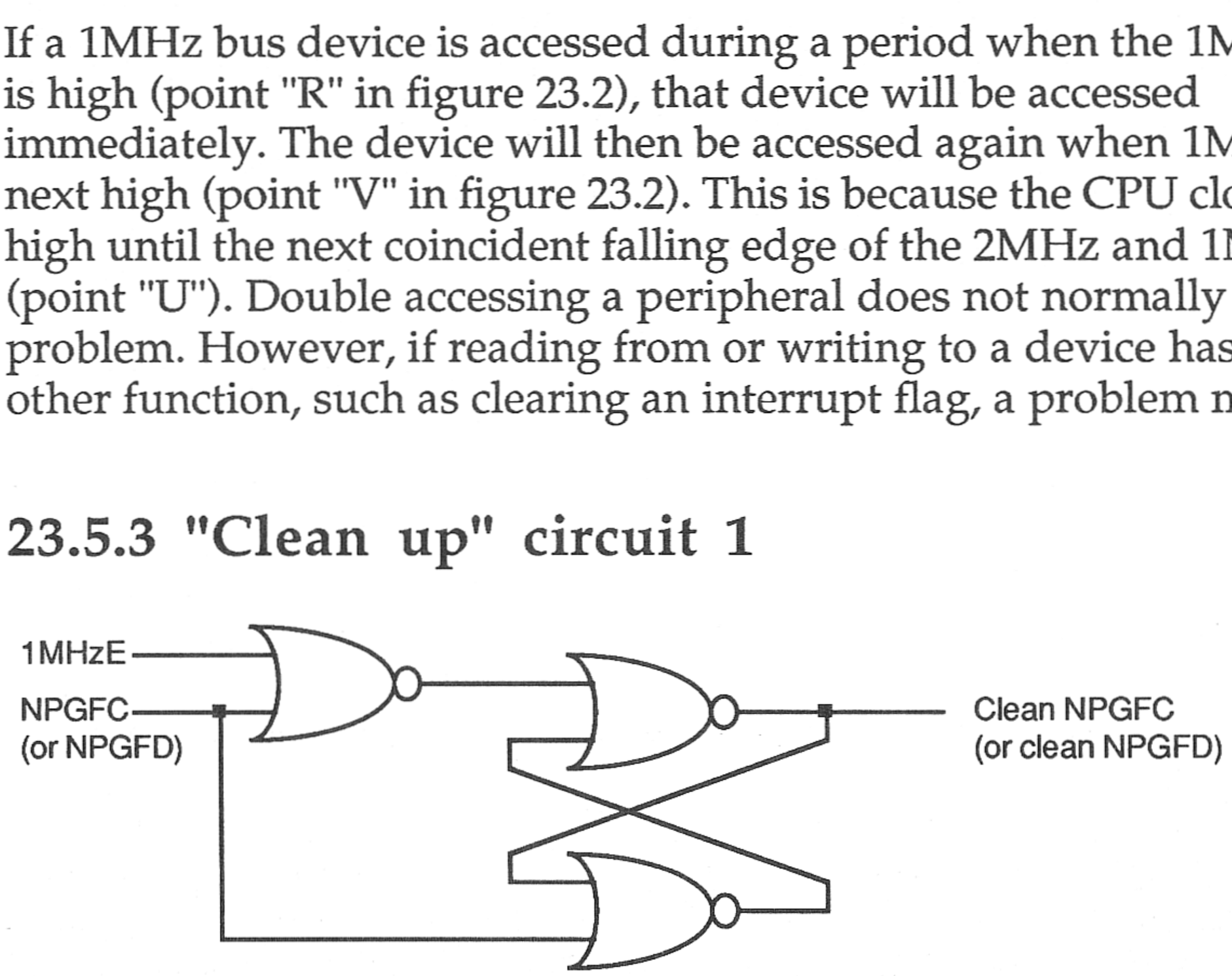
Addresses on the system address bus will only usually change when the 2MHz processor clock is low. However, the 1MHz clock is alternately

low, then high when the CPU addresses change. This gives rise to the address decoding glitches labeled "P" and "Q" in figure 23.2. The "Q" glitches are not normally important because the 1MHzE clock is then low. The "P" glitches can cause problems because the 1MHzE signal is then high. Spurious pulses may therefore occur on the various chip select pins, leading to possible malfunction of some devices.

23.5.2 Double accessing of 1MHz bus devices - PROBLEM 2

If a 1MHz bus device is accessed during a period when the 1MHzE clock is high (point "R" in figure 23.2), that device will be accessed immediately. The device will then be accessed again when 1MHzE is next high (point "V" in figure 23.2). This is because the CPU clock is held high until the next coincident falling edge of the 2MHz and 1MHz clocks (point "U"). Double accessing a peripheral does not normally present a problem. However, if reading from or writing to a device has some other function, such as clearing an interrupt flag, a problem may occur.

23.5.3 "Clean up" circuit 1



Circuit to remove glitches from NPGFC or NPGFD on 1MHz bus

Figure 23.3 - "clean up" circuit 1

The standard "clean up" circuit for the page select signals is shown in figure 23.3. Three NOR gates are used to create a standard R-S flip-flop with a gated input. The "clean page select" output (CNPGFC1) can only be set low if 1MHzE is low. The net effect of the circuit is illustrated in figure 23.2. Both of the problems outlined above are overcome, since the "P" glitches are removed and the page select only goes low at "S", after the 1MHzE clock has gone low. The "Q" glitches due to spurious addresses whilst 1MHzE is low are still present. In most applications, this will not affect circuit operation, but occasionally a totally glitch free page select will be required. Circuit 2 will provide this type of page select.

23.5.4 "Clean up" circuit 2

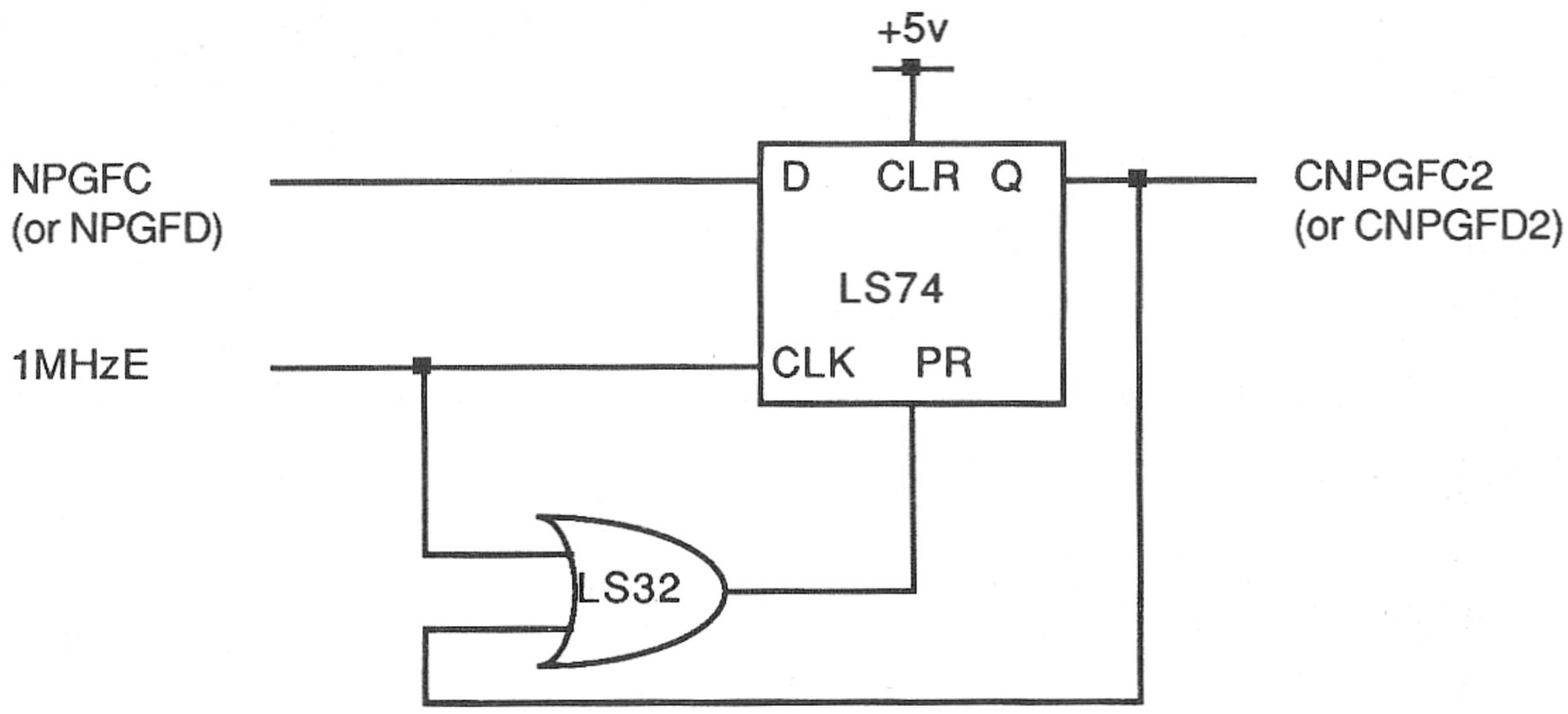


Figure 23.4 - "clean up" circuit 2

In situations in which a 100% "clean" page select signal is required, circuit 2 can sometimes be used. Before CNPGFC can go low, a valid page address with 1MHzE low must occur. The page low is then latched into a D-type flip-flop on the rising edge of the 1MHzE clock. As shown in figure 23.2, CNPGFC2 will go low a time flag (40nS) after 1MHzE goes high and it will remain valid until 40nS after 1MHzE has gone low again. Take care if any of the lines on the 1MHz bus are buffered, because delays introduced by buffers could make data invalid when it is latched. Refer to the precise timing information in section 23.6.5 for more details. Some peripheral circuits cannot be used with this clean-up circuit because the decoded chip select will not remain valid for a sufficient period.

23.6 Hardware requirements for 1MHz bus peripherals

All additional hardware designed to operate from the 1MHz bus must conform to the following standards:

23.6.1 Power supply

No power should be drawn from the BBC microcomputer. All peripherals should have their own integral power supply, or use a separate power supply unit.

23.6.2 Logic line loading

No more than one low power Schottky TTL load should be presented to any of the logic lines by a peripheral. In most instances, this means that all logic lines will have to be buffered for each peripheral.

23.6.3 Connection to the BBC microcomputer

Connection to the BBC microcomputer should be via a 600mm length of 34-way ribbon cable terminated with a 34-way IDC socket. The 1MHz bus connections should "feed through" the unit, i.e. a 34-way output header plug connector should be provided so that more devices can be connected as required.

23.6.4 Bus termination

All bus lines except NRST, NNMI and NIRQ should be provided with the facility for adding optional termination. The recommended way of terminating lines is to connect each one to +5V with a 2K2 resistor and to 0V with a 2K2 resistor.

23.6.5 Timing requirements

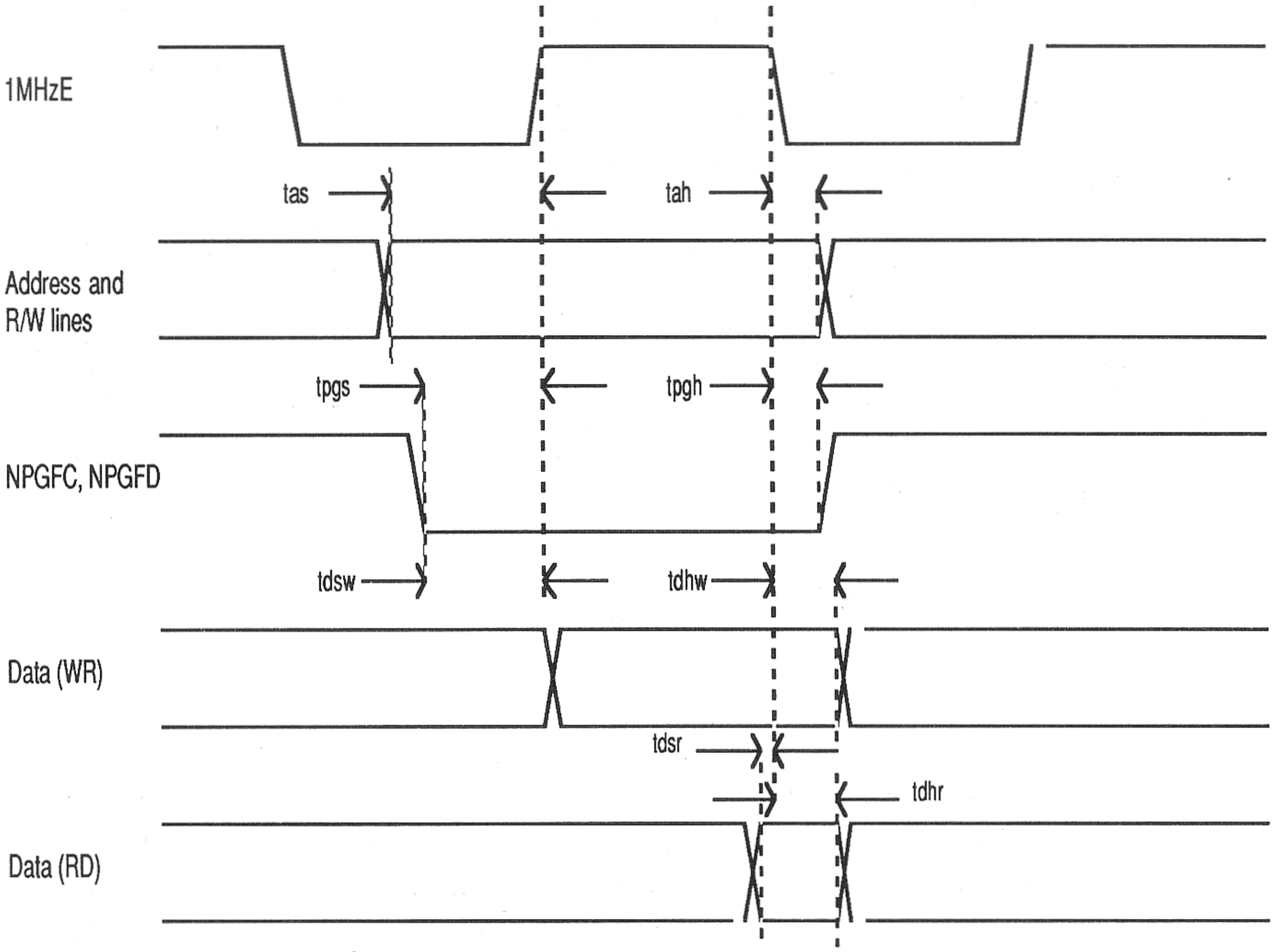


Figure 23.5 - 1MHz bus timing requirements

The 1MHz bus timing requirements are illustrated in the timing diagram, figure 23.5. It should be noted that these timings are based on the assumption that only one peripheral is attached to the bus. Heavier loading may extend the *rise* and *fall* times of 1MHzE with possible adverse effects on timings.

The timing requirements are:

Description	Symb.	Min.	Max.
Address (and Read/Write)			
Set-up time	t as	300nS	1000nS
Hold time	t ah	30nS	-
NPGFC & NPGFD			
Setup time	t pgs	250nS	1000nS
NPGFC & NPGFD			
Hold time	t pgh	30nS	-
Write data set-up time	t dsw	-	150nS
Write data hold time	t dhw	50nS	-
Read data set-up time	t dsr	200nS	-
Read data hold time	t dhr	30nS	-

23.7 Master, Compact & Electron Cartridge Interface

This interface was originally implemented on the Electron Plus 1 expansion unit. It allowed the Electron to accept modules containing paged ROMs (there is no facility on the main Electron pcb for plugging in such ROMs). Two similar ROM cartridge slots are provided on the Master 128. Typically these will be used with extra RAM or ROM plug-in modules. Additionally it is possible to plug fairly sophisticated pieces of peripheral hardware into these slots. The Master Compact cartridge interface has a similar set of signals, but the connector is slightly different and is not directly compatible with Electron or Master 128 cartridges.

Select 1MHz bus/cartridge OSBYTE

Call address &FFF4
Indirected through &20A
A=&6B(107)

Entry parameters:

X=0 selects the external bus running at 2MHz
X=1 selects the internal bus running at 1MHz
Y=0

On exit,

X is preserved
Y is corrupted

To facilitate the use of peripheral hardware devices, the normal '1MHz bus' signals are present on the cartridge connector together with several additional signals. It must be noted that the '1MHz bus' signals present on the cartridge interface actually operate at '2MHz' on the Master and Compact! On the Master it is necessary to select between the normal 1MHz interface (present on the connector below the keyboard) and the cartridge interface using OSBYTE &6B. The Compact does not have a normal 1MHz bus connector, so the cartridge is always selected.

Care must be taken when producing hardware for connection to the cartridge sockets. Many of the lines are not buffered, especially on the Electron and Master Compact.

23.7.1 Using 128K bytes of EPROM in Master ROM slots

The ROM cartridges are each designed to provide up to two normal 16K paged ROMs or RAMs in the Master's memory map. Typically one 32K byte EPROM (a 27256) will be used. The chip will be selected by the ROMOE signal (on pin A2) with the low order bit of the ROM paging register (QA on pin A16) being used to switch between the two halves of the memory in the chip.

To extend the memory which can be present in each slot, it is possible to use two 27513 chips. These EPROMs are each organised as 4 x 16K byte blocks. Only one of these four blocks is selected to appear in the chip at any instant. The current block is selected by writing the block select number 0 - 3 to any address within the EPROM, thereby loading the EPROM's internal 'block select register'. This will usually be done by code within the EPROM itself. For example, the following code changes between blocks within the EPROM:

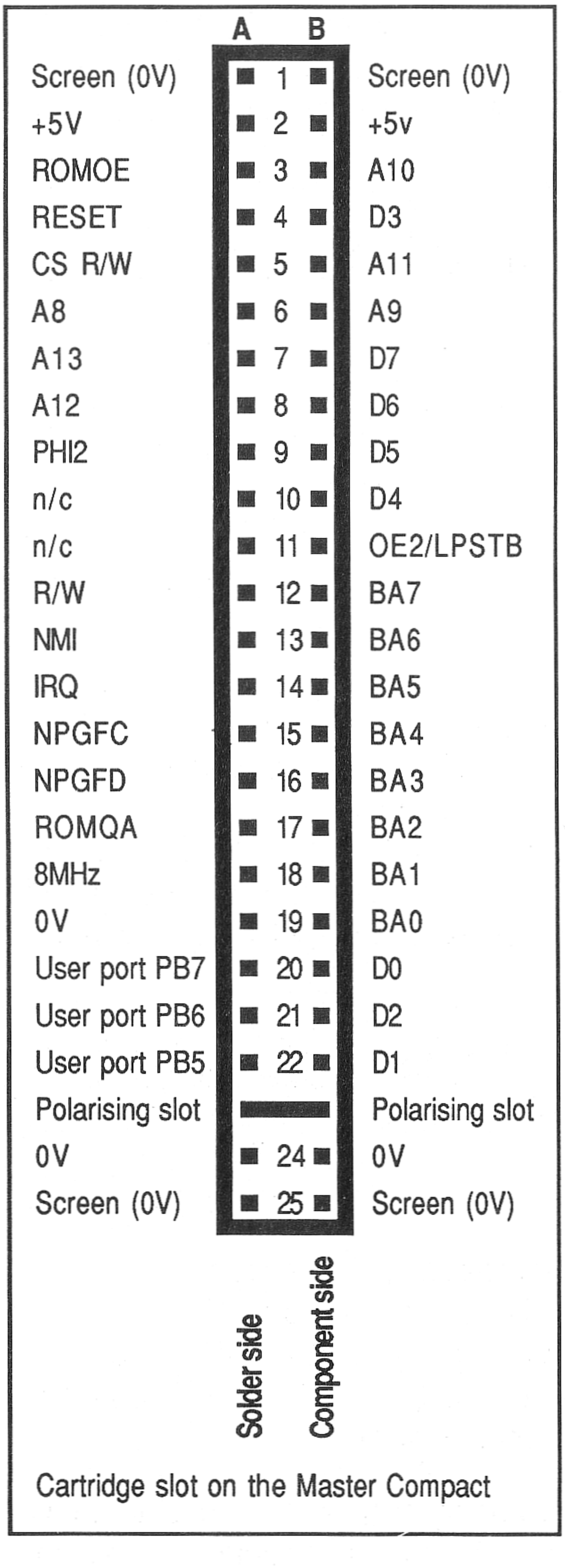
```
            LDA  #newblk \ this switching code is duplicated at
                         \ the same location in each 16K byte block
.blksw      STA  &8000   \ on entry A contains block to select.
                         \
                         \
```

To produce a paged ROM which resides in a 27513, interception of system calls to the ROM will normally occur in block 0. This is the 16K block which always appears at power-up. Once the ROM has detected that it should be selected, it can access the extra 48K of EPROM by switching between blocks in the manner described.

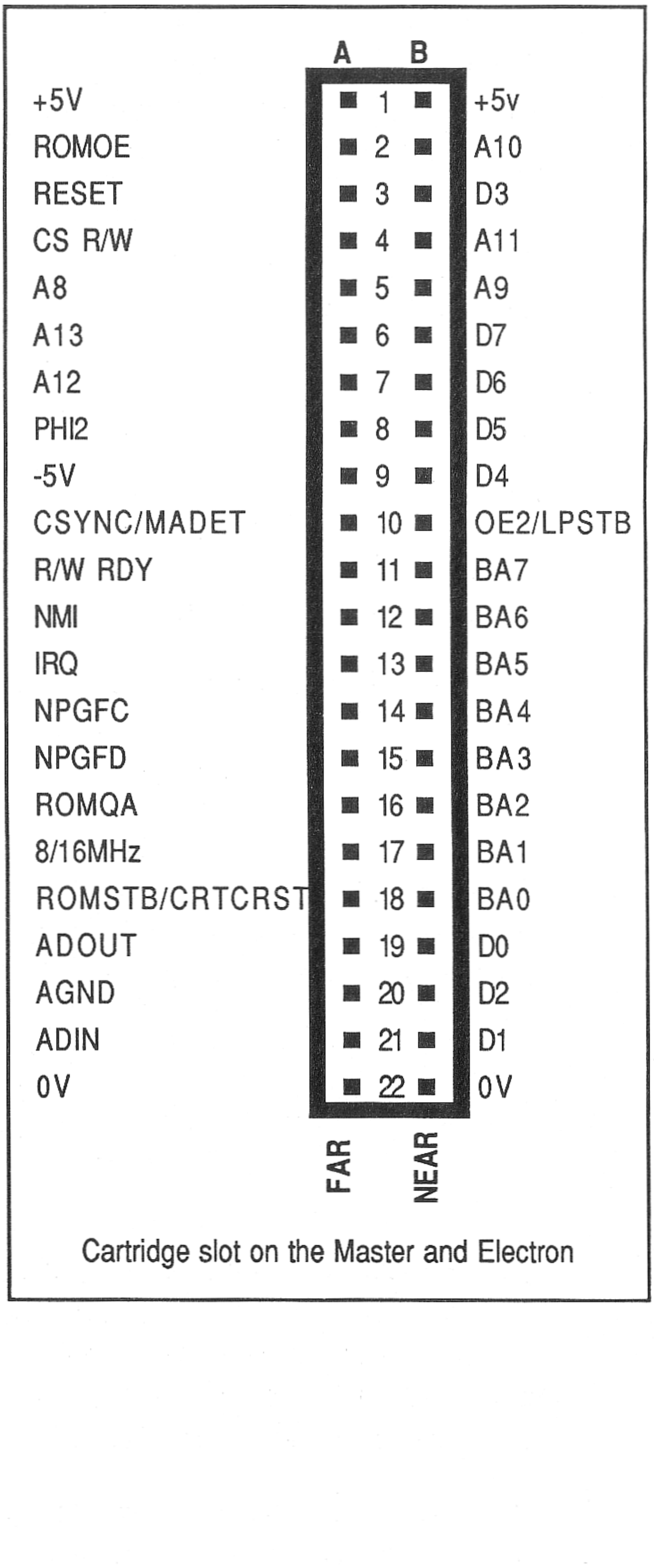
Interrupts can only be properly handled by duplicating the interrupt processing code, or by placing code at the start of blocks 1,2 and 3 to

switch to block 0 when an interrupt service call is made by the OS. A record of the currently selected page number in the EPROM should be kept in RAM. After processing any interrupt calls from the OS, the correct page in EPROM must be re-selected before returning.

23.7.2 Bus signal definitions



Cartridge slot on the Master Compact



Cartridge slot on the Master and Electron

The two types of ROM cartridge connectors are shown above. The current specifications for the signals on both the Master, Compact and Electron (as viewed from within the cartridge) are:

+5v	Connected to the system +5 volt supply. Up to 50mA can be drawn from a slot in an Electron Plus 1. Up to 150mA can be drawn from a slot in a Master fitted with internal co-processor and disc drives. On the Master Compact total current drain from the ROM cartridge connector, RGB connector and the joystick/mouse port should not exceed 200mA.
0v	Zero volts system earth return for digital circuits.
-5v	This negative 5 volt supply at up to 20mA may not be present on all Acorn cartridge interfaces, so to maintain compatibility negative voltages should be generated from the +5v supply using a small inverter. This is not connected on the Master Compact.
D0-D7	Data bus lines 0-7. TTL level input/output lines.
BA0-BA7	Address line inputs from A0-A7. On the Master these are buffered and are guaranteed to hold addresses valid for 125ns after PHI2 goes low. On the Electron these are un-buffered address lines.
A8-A13	Address line unbuffered inputs from system address bus.
PHI2	CMOS level input from computer's PHI2 clock.
ROMOE	Output enable used to switch on the output buffers of cartridge memory devices when low during the PHI2 period of the system clock, not guaranteed low at other times. Active low CMOS input. This can be used directly as the chip enable to 32K byte EPROMs, with QA used to select which half of the EPROM is accessed.
RST	System reset - active low CMOS input.

CSRW Read/write line input from the 6502 on the Electron. On the Master the function of this signal varies depending upon the area of memory being accessed. It is equivalent to the CPU read/write line during accesses to &FC00 - &FEFF during PHI2. For accesses to all other areas of memory it is an active high chip select for memory devices during PHI2, but is not guaranteed low at other times.

R/W or RDY On the Electron this is an open collector active low ready output to the CPU WAIT control. The CPU will extend its cycle when this signal is low, but only works with CMOS CPUs. On NMOS CPUs only the read cycles are extended. On the Master and Compact this is the R/W data direction control input. Cartridges are being written to when this TTL signal is low and they may drive the bus during PHI2 if this signal is high and the cartridge device is selected.

NMI Active low open collector Non-maskable interrupt output to the system NMI line.

IRQ Active low open collector Interrupt request output to the system IRQ line.

NPGFC Page &FC select input. On the Master this signal only becomes active when bit IFJ in ACCCON register is set (see section 12.2).

NPGFD Page &FD select input. See note for NPGFC signal.

ROMQA Memory paging select bit. This TTL level input is the least significant bit of the ROM select latch located at &FE30 in the Master and &FE05 in the Electron ULA.

CLOCK Electron 16MHz input clock. On the Master the operation is defined by links in the computer which select the signal as an 8MHz input to cartridge or 16MHz output to the computer system clock! (see Master reference manual 1). On the Compact this is an 8MHz input to the cartridge.

ROMSTB/ CRTCRST — On the Master CRTCRST is an active low output signal of CRTCRST, the system CRTC reset input; provided for genlock use. On the Electron this is an active low input which selects &FC73 and is intended for use as a paging register. It is connected to 0v on the Compact.

CSYNC/ MADET — This is not connected on the Electron and Compact. On the Master it has two possible functions, as defined by link 12 in the computer. MADET is the default link setting. It allows cartridges to detect which machine they are plugged into and is connected to 0v in the Master. With CSYNC selected (position B), the composite sync. signal (TTL level) is available for genlock use.

ADOUT — Audio output providing the sum of all audio inputs to the computer. No significant load should be taken from this pin.

AGND — Audio ground should be used as the ground line for audio circuitry instead of the system 0v line to minimise audio noise.

ADIN — Audio output from the cartridge to the computer.The computer's audio amplifier input impedance is at least 1K ohm and only one cartridge should use this at any time. On the Master, this may be used as an output from a speech system module fitted in the cartridge.

OE2/LPSTB — This provides a connection between the two cartridges on the Master if link 21 is removed. With the link in place it connects to the CRTC light pen strobe input and is pulled high by a resistor. On the Compact it connects to the CRTC lpstb input. On the Electron it provides an active low ROM enable for ROM number 13. It is low during the active low portion of PHI2 and is not guaranteed high at other times.

PB5-PB7 — Master Compact only. These are three lines connected to the User 6522 VIA. The other user port lines on the Compact are accessible on the joystick/mouse port.

Note: Acorn may change these specifications, so it is advisable to check with their technical support department before assuming that the exact operational details are as described above!

24 Miscellaneous topics

24.1 BREAK/reset associated calls

The following calls have functions which come into effect following a reset. A hard reset refers to the CTRL+BREAK key combination and a soft reset is caused by pressing the BREAK key alone. A power on reset is selected when the system 6522 interrupts are not enabled. A power on reset may be caused by masking the system 6522 interrupts before forcing a reset. This is done by writing the value &7F to SHEILA address &4E (i/o processor address &FE4E).

24.1.1 Read/write ESCAPE+BREAK effects OSBYTE call

Call address &FFF4
Indirected through &20A
A=&C8 (200)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag value is returned in X.

bit 0=0	Normal ESCAPE action
bit 0=1	ESCAPE disabled unless caused by OSBYTE &7D/125
bits 1 to 7=0	Normal BREAK action
bits 1 to 7=1	Memory cleared on BREAK

e.g. A value 0000001x (binary) will cause memory to be cleared on BREAK.

24.1.2 Read/write message suppression OSBYTE call

Call address &FFF4
Indirected through &20A
A=&D7 (215)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag value is returned in X.

bit 7 If clear then ignore OS startup message.
If set then print up OS startup message as normal.

bit 0 If set then if an error occurs in a !BOOT file in *ROM, carry on but if an error is encountered from a disc !BOOT file because no language has been initialised the machine locks up.
If clear then the opposite will occur, i.e. locks up if there is an error in *ROM.

This can only be over-ridden by a paged ROM on initialisation or by intercepting BREAK, see OSBYTE calls &F7 to &F9.

24.1.3 Read/write reset intercept code OSBYTE calls

Call address &FFF4
Indirected through &20A
A=&F7 (247)
A=&F8 (248)
A=&F9 (249)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

The contents of these locations must be JMP instruction for BREAKs to be intercepted (the operating system identifies the presence of an intercept by testing the first location contents equal to &4C -JMP). This code is entered twice during each break. On the first occasion C=0 and is performed before the reset message is printed or the Tube initialised. The second call is made with C=1 after the reset message has been printed and the Tube initialised.

24.1.4 Read type of last reset OSBYTE call

Call address &FFF4
Indirected through &20A
A=&FD (253)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old BREAK type is returned in X.

This location contains a value indicating the type of the last BREAK performed.

0 = soft BREAK
1 = power up reset
2 = hard BREAK

24.1.5 Read/write start-up option byte OSBYTE call

Call address &FFF5
Indirected through &20A
A=&FF (255)

This location is used similarly on the BBC microcomputer and the Electron.

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old start-up byte value is returned in X.

On the BBC microcomputer this location is determined by the 8 links on the right hand front corner of the keyboard pcb following a hard BREAK.

On the Electron there are no keyboard links, the default value of this location is &FF (255) and this OSBYTE is the only way of resetting the start up options. The bits referring to disc drive timings are not relevant on this machine.

bits 0 to 2	screen MODE selected following reset.(MODE number = 3 bit value)
bit 3	if clear reverse action of SHIFT+BREAK.
bits 4 and 5	used to set disc drive timings (see below)
bits 6 and 7	not used by operating system.(reserved for future applications)

Disc drive timing links :-

link 3	link 4	step time	settle time	head load
1	1	4	16	0
1	0	6	16	0
0	1	6	50	32
0	0	24	20	64

24.2 Printer OS calls

24.2.1 Select printer destination OSBYTE call

Call address &FFF4
Indirected through &20A
A=&05 (5)

Entry parameters:

X determines print destination

X=0 Printer sink (printer output ignored)
X=1 Parallel output
X=2 RS423 output (will act as sink if RS423 is enabled using OSBYTE with A=3)
X=3 User printer routine
X=4 Net printer
X=5-255 User printer routine

Default settings:

BBC Micro	*FX 5,1
Electron	*FX 5,0

On Exit:

A is preserved
X contains the previous setting
Y and C are undefined
Interrupts are enabled by this call
This call is not reset to default by a soft break

24.2.2 Read/write printer destination OSBYTE call

Call address &FFF4
Indirected through &20A
A=&F5 (245)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X. The contents of the next location are returned in Y.

This call is used by OSBYTE &5/*FX 5. Using this call does not check for the printer previously selected being inactive or inform the user printer routine.

24.2.3 Set printer ignore character OSBYTE call

Call address &FFF4
Indirected through &20A
A=&06 (6)

Entry parameters:

X contains the character value to be ignored

On exit:

A is preserved
X contains the previous setting
Y and C are undefined

24.2.3 Read/write printer ignore character OSBYTE call

Call address &FFF4
Indirected through &20A
A=&F6 (246)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old value is returned in X.

This call is used by OSBYTE &6.

24.2.4 User print vector, UPTV

Indirection address &222

A user print routine can be implemented by intercepting this vector. Whenever a change in printer type is made using OSBYTE &05 the print vector is called. A user print routine should respond when printer type 3 is called.

The operating system will activate the user printer routine and thereafter call it regularly at intervals of 10 milliseconds. Characters will be placed in the printer buffer, and it is up to the user printer routine to remove characters and send them to the printer hardware. When the printer routine finds that the buffer is empty it should then declare itself inactive. The operating system will then re-activate the routine when characters start entering the buffer again.

The user printer driver should preserve all registers and return via the old UPTV value.

On entry:

X contains the buffer number to be used
Y contains the printer number (i.e. the *FX 5 value)

N.B. The routine should only respond if it recognises the printer number as its own.

The accumulator contains a reason code for the call:

A=0
When the printer driver is active the operating system makes this call every 10 ms. The printer driver should examine its hardware and if it is ready for another character should remove a character from the assigned buffer and send it to the printer. A call to the REMV vector should be made to obtain the character (see section 6.9.2). When the printer driver has emptied the printer buffer it should then declare itself inactive by making an OSBYTE call &7B. This will allow the user to select a new printer driver using OSBYTE &5, will stop further calls with A=0 and thereafter when the printer buffer is used again will cause a call with A=1 to be made (see below).

A=1
When a printer driver is inactive this call is made to tell the routine that the printer buffer is no longer empty and the printer driver should now become active. If the printer driver is able to become active it should remove a character from the assigned buffer. If the buffer is not empty it should return with the carry flag clear to indicate that it is now active.

Having thus signalled itself as active, the printer driver will receive the 10 ms calls with A=0.

A=2
When the VDU drivers receive a VDU2, this call is made. Characters may be printed even when this control character has not been received if certain *FX3 options are selected.

A=3
This call is made when a VDU3 is received.

A=5
The selection of a new printer driver will cause this call to be made to the printer vector. Any OSBYTE &5 call causes this call to be made.

24.2.5 Printer driver going dormant OSBYTE call

Call address &FFF4
Indirected through &20A
A=&7B(123)

Entry parameters:
X should contain the value 3 (printer buffer i.d.)

This OSBYTE call should be made by user printer drivers when they go dormant. The operating system will need to wake up the printer driver if more characters are placed in the printer buffer.

On exit:
A and X are preserved
Y is preserved in the i/o processor
(N.B. but not passed across the Tube)
C is undefined

24.3 *CODE and *LINE

***CODEx,y (*CO.)**
OSBYTE &88

Call address &FFF4
Indirected through &20A
A=&88(136)

Entry parameters:
X and Y registers passed on to user vector (USERV)

This command enables the user to incorporate his own command into the operating system command table. *CODE executes machine code indirected through the user vector (USERV) at locations &200,&201 (low-byte, high-byte). The default contents of the user vector produce the "Bad Command" message. The machine code at USERV is entered with A=0, X=x and Y=y.

For example:

```
10 DIM MC% 100
20 OSASCI=&FFE3
30 USERV=&200
40 FOR opt%=0 TO 3 STEP3
50   P%=MC%
60   [
70   OPT opt%
80   .write
90   CMP #0              \ *CODE or *LINE ?
100  BEQ code            \ if *CODE call act upon it
110  BRK                 \ if *LINE call print error message
120  NOP
130  ]
140  $P%="*LINE not enabled"
150  P%=P%+LEN$P%
160  [
170  BRK
180  .code TXA           \ transfer contents of X reg. to Acc.
190  JSR OSASCI          \ print ASCII character
200  RTS                 \ return to BASIC
210   ]
220   NEXT
230 ?USERV=write MOD 256
240 ?(USERV+1)=write DIV 256
```

This example prints out the ASCII character corresponding to the value of the first parameter given to the *CODE command. After this program has been run typing in "*CODE 65" or "*FX 136,65" will cause a letter "A" to be printed. The second parameter (stored in Y) if included, is ignored.

***LINE<text> (*LI.)**

This command executes machine code at the location pointed to by the contents of the user vector (USERV) at locations &200,&201 (low-byte,high-byte). The command enters this code with the A=1, X=least significant byte of string address and Y=most significant byte of string address. *LINE provides an easy method of incorporating a user function into the operating system command table.

e.g.

```
10 DIM MC% 100
20 OSASCI=&FFE3
```

```
 30 USERV=&200
 40 FOR opt%=0 TO 3 STEP3
 50   P%=MC%
 60   [
 70   OPT opt%
 80   .write
 90   CMP #1              \ *CODE or *LINE?
100   BEQ code            \ execute machine code if *LINE
110   BRK                 \ otherwise print out error message
120   NOP
130   ]
140   $P%="*CODE not enabled"
150   P%=P%+LEN$P%
160   [
170   BRK
180   .code STX &70       \ *LINE code entry point and store
181   STY &71             \ string address: low-byte,high-byte
182   LDY #0                 \ set up Y register for indexing
183   .loop LDA (&70),Y \ Post-Indexed Indirect addressing
190   JSR OSASCI          \ print out character
191   INY                 \ increment index
192   CMP #&0D            \ test for end of string
193   BNE loop            \ if not last character go round again
200   RTS                 \ finished
210   ]
220   NEXT
230 ?USERV=write MOD 256
240 ?(USERV+1)=write DIV 256
```

24.4 Miscellaneous OSBYTE calls

This section contains a collection of miscellaneous operating system calls.

24.4.1 Identify Machine/OS version OSBYTE calls

OSBYTE &00 - Read OS version number

Call address &FFF4
Indirected through &20A
A=0

Entry parameters:

X=0	Execute BRK, print OS version
X<>0	RTS with OS version returned in X

On exit:

X=0, OS 1.00 (Early BBC B or Electron OS 1.00)
X=1, OS 1.20 or American OS
X=2, OS 2.00 (BBC B+)
X=3, OS 3.2 (Master 128)
X=4, OS 4.0 (Master ET)
X=5, OS 5.0 (Master Compact)
A and Y are preserved
C is undefined

OSBYTE &81 - Read machine type

Call address &FFF4
Indirected through &20A
A=81

Entry parameters:

X=0	Execute BRK, print OS version
Y=&FF	RTS with OS version returned in X

On exit:

A=0	BBC microcomputer OS 0.10
A=1	Acorn Electron OS 1.00
A=&FF	BBC microcomputer OS 1.00 or 1.20
A=&FE	BBC microcomputer OS 1.10 (USA)
A=&FD	Master 128 OS 3.20
A=&FC	BBC microcomputer OS 1.20 (West Germany)
A=&FB	BBC B+ OS 2.00
A=&FA	Acorn ABC OS
A=&F5	Master Compact OS 5.10

OSBYTE call &F0 - Read country code

Call address &FFF4
Indirected through &20A
A=&F0 (240)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The country code value is returned in X.

This location contains a value indicating the country for which this version of the operating system has been written.

country code	country
0	United Kingdom
1	United States

24.4.2 Set user flag OSBYTE call

Call address &FFF4
Indirected through &20A
A=1

Entry parameters:
The user flag is replaced by X

On exit:
X=old value

Read/write user flag OSBYTE call

Call address &FFF4
Indirected through &20A
A=&F1 (241)

<NEW VALUE>=(<OLD VALUE> AND Y) EOR X

The old flag value is returned in X.

This location is reserved as a user flag for use with *FX 1.

Default value 0.

Glossary

Address bus - 16 electrical connections between the CPU and memory. Each line can be at logic 0 or 1, allowing 2^{16} (65536) different locations.

Active low - signals are valid when they are at logic level 0.

ADC (Analogue to digital converter) - an electronic circuit which can accept an analogue voltage and provide a digital output of that voltage.

Asynchronous - two devices operate asynchronously when they both operate independently.

Baud rate - defines the rate at which a serial data link transfers data. One baud is equal to one bit of data transferred per second.

Bidirectional- a communication line is bidirectional if data can be transmitted and received over it. The databus is bidirectional.

Bit of memory - this is the fundamental unit of a computer's memory. It may only be in one of two possible states, usually represented by 0 or 1.

Buffer - a software buffer is an area of memory set aside for data in the process of being transferred from one device or piece of software to another.

Byte of memory - 8 bits of memory. Data is normally transferred between devices one byte at a time over the data bus.

CMOS (Complementary metal oxide semiconductor) - a type of chip manufacturing technology which produces low power chips. The CMOS RAM and real time clock can therefore operate from a small battery for long periods.

CPU (Central processing unit) - the 6502 in the Acorn-BBC computers is the chip which does all the computing work associated with running programs.

Data bus - a set of eight connections over which all data transactions between devices are sent.

EEPROM (Electrically erasable programmable memory) - as used in the Master Compact to store variables without power applied to the memory chip. Unlike the Master 128 CMOS RAM, no battery is needed.

EPROM (Erasable programmable memory) - will retain data written into the device by a special programming unit even when power is

switched off. Erroneous data can be removed by exposing the chip to UV light, allowing the device to be re-programmed.

File handle - a single byte value which uniquely identifies an open file.

Firm keys - only on the Electron. The combination of pressing the CAPS LK/FUNC keys with other keys on the keyboard produces the codes for the normal function keys (which aren't present on the Electron), and produces a range of text strings when pressed.

Handshaking- this type of communications protocol is used when data are being transferred between two asynchronous devices. The essential feature of the handshaking signal is to inform the sender of data when the receiver of data is able to accept data.

High - sometimes used to designate logic level '1'.

Latch - is used to retain data applied to it after that data has been removed. It is rather like a memory location, with the output from the bits within the location being connected to some hardware.

LED (Light emitting diode) - an electronic device which emits light.

Low - sometimes used to designate logic '0'.

Mnemonic - the name given to the text string which defines a particular 6502 operation in the BASIC assembler. LDA is a mnemonic which means *load accumulator*.

MOS (Machine operating system) - see OS (operating system).

Nibble of memory - 4 bits of memory.

Opcode - the name given to the binary code of a 6502 instruction. For example, &AD is the hex opcode which means *load accumulator.*

Open collector - this is a characteristic of a transistor output. It simply means that the collector pin of the transistor is not driving a resistive load, ie. it is open.

OS (Operating system) - the machine code program resident in ROM which controls all the basic functions for the computer. These include. handling input and output to/from the screen, memory, discs, keyboard etc. and handling all low level functions like interrupts.

Page - a page of memory in the 6502 memory map is &100 (256) bytes long. There are 256 pages in the 65536 byte memory map.

Parallel - data transfers occur along several lines simulataneously.

Polling - a method of interrogating devices to check if they need to be serviced.

RAM (Random access memory) - memory which can be both written to and read from.

Rollover - allows two keyboard keys to be pressed simultaneously. The first key is in the process of having a finger removed, whilst the next is having a finger applied.

ROM (Read only memory) - memory which can only be read from, but not written to.

Serial - data transmitted along only one line is transmitted serially, ie. one bit at a time.

Stack - page &01 of memory used for temporary data storage. Data is pushed onto the stack in sequence, then removed by pulling it off. The last byte to be pushed is the first byte to be pulled. The stack is mainly used to store return addresses from subroutines.

TTL (Transistor transistor logic) - a type of chip manufacturing technology used in many small scale chips.

ULA (Uncommitted logic array) - a silicon chip which has been specially *committed* by Acorn to perform a particular function.

Word of memory - two bytes of memory, ie. 16 bits.

Bibliography

6502 Assembly Language Programming, L.A.Leventhal, OSBORNE/Mc Graw Hill, Berkeley, California

6850 Asynchronous Communications Interface Data Sheet, Thomson Semiconductors, 1981

Advanced User Guide for the Acorn Electron, Dickens & Holmes, Adder Publishing Ltd., 1984

Advanced User Guide for the BBC Microcomputer, Bray, Dickens & Holmes, Cambridge Microcomputer Centre, 1983

BASIC ROM User Guide for the BBC Micro, M.D.Plumbley, Adder Publishing Ltd., 1984

BBC Microcomputer system User Guide, John Coll, BBC Publications 1981, revised by Acorn Computers, 1985

Disc System User Guide, Brian Ward, BBC Publications, 1982

Master Operating System, David Atherton, Dabs Press, 1987

Master Reference Manual part 1, Acorn Computers Ltd., 1986

Master Reference Manual part 2, Acorn Computers Ltd., 1986

Programming the 6502, Rodnay Zaks, Sybex, 1980

Speech System User Guide, Acorn Computers Ltd., 1983

*CODE, 425
*FX calls: OSBYTE, 110
*LINE, 426
*ROM data format, 317
*TV command., 184
1MHz bus, 402
1MHz bus/cartridge selection, 413
2's Complement, 15
65c12: addressing modes, 25, 27
    BCD, 18
100Hz paged ROM polling system, 325
1770 Control Registers, 272
6502 second processor, 331
6522 IRQ bit mask OSBYTE, 130, 131
6522 VIAs, 380
6845 CRTC, 188
6850 control register, 244
6850 IRQ bit mask OSBYTE, 131
32016 second processor, 350
76489 sound chip, 375
Absolute addressing, 20
Absolute,X or Y addressing, 22
ACCCON, 162
Accumulator - A, 28
Accumulator addressing, 19
Active low, 430
ADC system, 365
ADC, 32
    conversion complete event, 120
ADC, 430
Add with Carry, 32
Address bus, 430
Addressing Modes, 19
Analogue to digital converter, 365, 430
AND, 33
Animation, 202
Arithmetic Shift Left, 34
ASCII character values, 165
ASL, 34
assembler, 6
    BRK handling, 12
    CALL and USR, 12
    conditional assembly + macros, 13
    delimiters, 7
    EQUates, 10
    labels, 10
    location counter (P%), 9
    mnemonics, 29
    option, 8
    two pass, 10
    zero page, 14
Asynchronous, 430
BASIC ROM number, 324
Baud rate setting, 242
Baud rate, 430
BBR**, 35
BBS**, 36
BCC, 37
BCD, 17
    65C12, 18
BCS, 38
BEQ, 39
Bidirectional, 430
binary coded decimal, 17
Bit of memory, 430
BIT, 40
BMI, 41
BNE, 42
BPL, 43
BRA, 44
Branch always, 44
Branch if negative flag set, 41
Branch if overflow clear, 46
Branch if overflow set, 47
Branch on Bit Reset, 35
Branch on Bit Set, 36
Branch on Carry Clear, 37
Branch on Carry Set, 38
Branch on positive result, 43
Branch on result not zero, 42
Branch on result zero, 39
Break flag - B, 29
BREAK, 419
BRK, 12, 45
    associated OS calls, 132
    BRK vector, 132
    example BRK handling routine, 133
    interrupt handling, 125
BRKV (&202), 132
Buffer, 430
Buffers control and management, 136
buffers: example program, 139
    id numbers, 136
    using the buffer vectors, 138
BVC, 46
BVS, 47
Byte of memory, 430
CALL, 12
Carry flag - C, 28
Cartridge Interface, 413
Casette/RS423 selection, 246
Cassette system interrupt use, 126
Character input, 105
Character output, 103
Character set, 165
Characters, user defined, 171
CLC, 48
CLD, 49
Clear carry flag, 48
Clear decimal flag, 49
Clear interrupt disable flag, 50
Clear memory, 51, 91
Clear the overflow flag, 52
CLI, 50
Clocks, 352
CLR, 51
CLV, 52
CMOS RAM, 352
CMOS RAM/EEPROM, 356
CMOS Real Time Clock, 353
CMOS, 430
CMP, 53

CNPV, 137
Colours: logical & actual, 207
Compare memory and accumulator, 53
Compare memory with X register, 55
Compare memory with Y register, 56
Conditional assembly, 13
Converting files to *ROM format, 314
Count/purge buffer vector, 137
country code, 428
CP/M disc format, 349
CPU, 430
CPX, 55
CPY, 56
CTRL G, 374
Cursor width control, 206
cursor editing, 225
Cursor, 195
Data bus, 430
DEC/A*, 57
Decimal mode flag - D, 29
Decrement memory by one, 57
Decrement the Y register by one, 59
Decrement X register by one, 58
DEX, 58
DEY, 59
EEPROM, 430
Electron external sound, 375
Electron firm keys, 231, 292
Electron ULA IRQ mask OSBYTE, 132
ENVELOPE command, 372
EOR, 60
EPROM, 430
ESCAPE character setting, 232
ESCAPE, 419
escape: escape detected event, 121
    related OS calls, 147
Events and second processors, 333
Events, 119
    enable/disable OSBYTEs, 121
    event generation call, 113
    event vector, EVNTV, 119
    example program, 122
EVNTV, 113, 119
Examine buffer status OSBYTE, 145
Exclusive OR, 60
Exploding character definitions, 171
Extended Vectors, 312
File handle, 431
Filing System Calls, 112
Filing system: RAM control, 164
Filing systems, 250
    *ROM, 264
    cassette, 261
    disc, 264
    IEEE488, 269
    network, 268
    telesoft, 269
Firm keys on Electron, 231
Firm keys, 431
flashing colours, 178
Floppy Disc Hardware, 270
Flush specific buffer OSBYTE, 143
Forced interrupt, 45
FRED, 153
FRED, 403
Function keys, 225
FX command: OSBYTE, 110
Get character from buffer OSBYTE, 145
graphics cursor position, 175
GSINIT, 107
GSREAD, 108
Handshaking, 431
High, 431
I/O routines, 103
Immediate addressing, 19
Implicit addressing, 19
INC/A, 61
Increment memory by one, 61
Increment the Y register by one, 63
Increment X register by one, 62
Indexed Addressing, 22
Indirect addressing, 21
INKEY, 221
Input stream selection, 241
input buffer: buffer full event, 120
    character entry event, 120
    insert character OSBYTE, 146
input stream, 105
Insert value into buffer OSBYTE, 144
insert value into buffer vector, 136
Installing ROMs in the B Plus, 289
Installing ROMs in the Master 128, 290
Installing ROMs in the Master Compact, 291
Installing ROMs in the model B, 288
INSV, 136
Intercepting interrupts, 129
Interlace, 193
Interrupt disable - I, 28
Interrupt Request Vector, 129
Interrupts and second processors, 335
Interrupts, 124
    interrupt interception, 129
    OS interrupt handling routine, 125
Interval timer, 352
interval timer: event, 120
INX, 62
INY, 63
IRQ1V (&204), 125
IRQ1V, 129
IRQ2V, 129
JIM, 153
JIM, 403
JMP, 64
joystick, 365, 371
JSR, 65
Jump Subroutine, 65
Jump to new location, 64
Key values, 219
Keyboard scan, 221
keyboard auto-repeat delay, 224
keyboard disable, 228
Language ROMs, 291

language ROM number, 324
Languages in ROM, 284
LDA, 66
LDX, 67
LDY, 68
LED, 431
Light pens, 196
Load accumulator from memory, 66
Load X register from memory, 67
Load Y register from memory, 68
Logical AND, 33
Logical Shift Right, 69
Low, 431
LSR, 69
Machine Code Arithmetic, 15
Machine operating system, 431
machine code, 3
Macros, 13
Maskable Interrupts, 125
memory, 155
    access control register, 162
    calls concerning memory use, 157
    OS allocation + use, 114
    shadow, 158
Mnemonic, 431
mnemonic, 30
MOS, 431
Negative flag - N, 29
network: error event, 121
Nibble of memory, 431
NMI, 125
No operation, 70
Non Maskable Interrupts, 125
Non-vectored OSRDCH, 106
Non-vectored OSWRCH, 103
NOP, 70
Opcode, 431
Open collector, 431
Operating System Calls, 102
Operating system, 431
operating system: use of memory, 114
    zero page use, 114
OR memory with accumulator, 71
ORA, 71
OS startup message, 420
OS version, 427
OS, 431
OSASCI, 104
OSBYTE, 109
    &00 identify OS version number, 427
    &01 set user flag, 429
    &02 select input stream, 105
    &02 select input stream, 241
    &03 select output stream, 108
    &03 select output stream, 241
    &04 enable/disable cursor editing, 225
    &05 select printer destination, 422
    &06 set printer ignore character, 423
    &07 set receive baud rate, 242
    &08 set transmit baud rate, 242
    &09 set flash mark duration, 178
    &0A set flash space duration, 179
    &0B set keyboard auto-repeat delay, 224
    &0C set keyboard auto-repeat rate, 224
    &0D disable events, 121
    &0E enable events, 121
    &0F flush selected buffer(s), 143
    &10 select ADC channel, 366
    &11 force ADC conversion, 367
    &12 reset function keys, 227
    &13 wait for vertical sync., 182
    &14 explode character definition RAM, 172
    &14 restore default font definitions, 172
    &15 flush specific buffer, 143
    &16 increment polling semaphore, 325
    &17 decrement polling semaphore, 325
    &18 select external sound (Electron), 375
    &19 restore default font definitions, 173
    &44 Test for sideways RAM, 159
    &54 sideways RAM allocation, 160
    &6B select 1MHz bus/cartridge, 413
    &6C direct screen memory for access, 157
    &6D make temporary fs permanent, 260
    &70 select memory for VDU, 158
    &71 select memory for display, 158
    &72 write shadow memory use, 159
    &73 blank or restore palette (Electron), 181
    &74 reset sound system (Electron), 375
    &75 read VDU status, 176
    &76 make LEDs show keyboard status, 228
    &77 close *SPOOL/*EXEC files, 257
    &78 write current keys pressed, 222
    &79 keyboard scan, 221
    &7A keyboard scan from &10, 222
    &7B printer driver going dormant, 425
    &7C clear ESCAPE condition, 147
    &7D set ESCAPE condition, 147
    &7E clear ESCAPE + effects, 147
    &7F check for EOF, 258
    &80 read ADC channel, 365
    &80 read buffer status, 143
    &81 read key with time limit, 221
    &81 read machine type, 428
    &82 read high order address, 257
    &83 read OSHWM, 117
    &84 return HIMEM, 157
    &85 return HIMEM for a given mode, 157
    &86 read i/p cursor position, 175
    &86 read o/p cursor position, 174
    &87 read chr and screen mode, 174
    &88 *CODE, 425
    &89 *MOTOR, 262
    &8A insert value into buffer, 144
    &8B *OPT, 258
    &8C *TAPE, 262
    &8D *ROM, 264
    &8E select language ROM, 322
    &8F issue ROM service call, 322
    &90 *TV command, 184
    &91 get character from buffer, 145
    &92 read from FRED, 403
    &93 write to FRED, 403

&94 read from JIM, 403
&95 write to JIM, 403
&96 read from SHEILA, 403
&97 write to SHEILA, 403
&98 examine buffer status, 145
&99 put character into i/p buffer, 146
&9A reset flash cycle (Electron), 180
&9A write video ULA & OS copy (B/M), 180
&9B write ULA palette & OS copy, 181
&9C r/w 6850 control register, 244
&9D fast tube BPUT, 254
&9E read from speech processor, 374
&9F write to speech processor, 374
&A0 read VDU variable value, 183
&A1 read CMOS RAM/EEPROM, 356
&A2 write to CMOS RAM/EEPROM, 358
&A4 check for 6502 code, 326
&A6-&A7 r/w address of OS variables, 118
&A8 read lo address ROM pointer table, 323
&A9 read hi address ROM pointer table, 323
&AA read lo address ROM info. table, 323
&AB read hi address ROM info table, 323
&AC read key translation table lo addr., 223
&AD read key translation hi address, 223
&AE read VDU variables lo address, 183
&AF read VDU variables hi address, 183
&B0 r/w CFS timeout counter, 249
&B1 read input source, 105
&B2 r/w keyboard semaphore, 230
&B3 read/write primary OSHWM, 117
&B4 read/write OSHWM, 117
&B5 r/w RS423 mode, 243
&B6 read character explosion state, 173
&B7 CFS/RFS switch, 263
&B8 read ULA control OS copy, 182
&B9 r/w polling semaphore, 326
&B9 read ULA palette OS copy, 182
&BA read active ROM at last BRK, 135
&BB read BASIC ROM number, 324
&BC read current ADC channel, 367
&BD read maximum ADC channel, 367
&BE r/w ADC conversion type, 368
&BF r/w RS423 use flag, 244
&C0 read 6850 OS copy, 245
&C1 r/w flash counter, 180
&C2 r/w flash mark duration, 179
&C3 r/w flash space duration, 179
&C4 r/w keyboard auto-repeat delay, 224
&C5 r/w keyboard auto-repeat rate, 225
&C6 r/w *EXEC file handle, 258
&C7 r/w *SPOOL file handle, 258
&C8 r/w ESCAPE+BREAK effects, 419
&C8 r/w ESCAPE/BREAK effect, 148
&C9 r/w keyboard disable, 228
&CA r/w keyboard status, 229
&CB r/w Electron IRQ bit mask, 132
&CB r/w RS423 handshake level, 246
&CC r/w firm key pointer (Electron), 231
&CC r/w RS423 input suppression flag, 246
&CD r/w firm key string len. (Electron), 231
&CD r/w RS423/cassette flag, 246
&CE-&D0
&D1 r/w speech suppression flag, 373
&D2 r/w sound suppression flag, 373
&D3 set BELL channel, 374
&D4 set BELL amplitude/envelope, 374
&D5 set BELL frequency, 374
&D6 set BELL duration, 374
&D7 r/w OS startup message suppression, 419
&D8 r/w soft key string length, 227
&D9 r/w lines since page halt, 176
&DA r/w VDU queue, 168
&DB r/w TAB key character, 231
&DC r/w ESCAPE character, 148
&DC r/w ESCAPE character, 232
&DD r/w chrs &C0-F status, 226
&DE r/w chrs. &D0-F status, 226
&DF r/w chrs. &E0-F status, 226
&E0 r/w chrs. &F0-F status, 226
&E1 r/w fn. key status, 226
&E2 r/w shift + fn. key status, 226
&E3 r/w ctrl + fn. key status, 226
&E4 r/w ctrl+shift+fn. key status, 226
&E5 r/w ESCAPE key status, 148
&E6 r/w ESCAPE effects, 149
&E7 r/w user via IRQ bit mask, 130
&E8 r/w 6850 IRQ bit mask, 131
&E8 r/w 6850 IRQ mask, 247
&E8 r/w ext. sound semaphore (Electron), 375
&E9 r/w system via IRQ mask, 131
&EA r/w Tube flag, 347
&EB test presence of speech processor, 373
&EC r/w output stream flag, 109
&EC select output stream, 242
&ED r/w cursor editing status, 225
&EE set base for numeric keypad, 230
&F0 read country code, 428
&F1 r/w user flag, 429
&F2 read serial ULA OS copy, 247
&F3 read timer state, 353
&F4 r/w soft key consistency flag, 227
&F5 r/w printer destination, 423
&F6 r/w printer ignore character, 423
&F7 r/w reset intercept code, 420
&F8 r/w reset intercept code, 420
&F9 r/w reset intercept code, 420
&FA r/w flag used by osbyte &70, 158
&FB r/w flag used by osbyte &FB, 159
&FC read language ROM number, 324
&FD read last BREAK type, 420
&FE r/w shift key effect, 230
&FE RAM size, 160
&FF r/w start-up option byte, 421
r/w OS call interception status, 118
unknown, 111
OSCLI, 112
OSEVEN, 113, 122
OSNEWL, 104
OSRDCH, 105
OSRDRM, 113
OSRDRM, 321
OSRDSC, 113

OSWORD, 109
&00 read string, 106
&01 read system clock, 352
&02 write system clock, 352
&03 read interval timer, 352
&04 write interval timer, 352
&05 read i/o processor memory, 347
&06 write i/o processor memory, 347
&07 SOUND command, 372
&08 ENVELOPE command, 372
&09 read pixel value, 175
&0A read character definition, 171
&0B read palette, 178
&0C write palette, 177
&0D read previous graphics position, 175
&0E read real time clock, 353
&0F write real time clock, 355
OSWRCH, 103
OSWRSC, 113
Output stream selection, 241
output buffer: buffer empty event, 120
output stream, 108
Overflow flag - V, 29
Page, 431
Paged ROMs, 283
Paged ROMs: hardware control, 161
palette, 177, 207
PANDORA., 351
PANOS operating system, 351
Parallel, 431
PHA, 72
PHP, 73
PHX, 74
PHY, 75
PLA, 76
PLOT numbers, 169
PLP, 77
PLX, 78
PLY, 79
Polling, 432
Post-indexed indirect addressing, 24
Pre-indexed absolute indirect addressing, 25
Pre-indexed indirect addressing, 23
Printer driver going dormant, 425
Printer ignore character, 423
Printer OS calls, 422
Printer VIA, 381
program counter: 6502, 29
Pull accumulator off stack, 76
Pull status register off stack, 77
Pull X from stack, 78
Pull Y from stack, 79
Push accumulator onto stack, 72
Push Status register onto stack, 73
Push X register onto stack, 74
Push Y register onto stack, 75
RAM Access Control, 161
RAM, 432
RAM: size (BBC model A or B), 160
Read buffer status OSBYTE, 143
Read character from, 108
Read line from input OSWORD, 106
Relative addressing, 24
Remove value from buffer vector, 137
REMV, 137
Return from Interrupt, 82
Return from subroutine, 83
ROL, 80
Rollover, 432
ROM paging register, 161
ROM, 432
ROM/RAM installation, 287
ROR, 81
Rotate one bit left, 80
Rotate one bit right, 81
RS232, 233
RS423 handshake level, 246
RS423 system interrupt use, 126
RS423/cassette selection, 246
RS423: error event, 121
RTI, 82
RTS, 83
SBC, 84
Screen mode memory maps, 210
Screen state, reading, 174
screen format, 189
Scrolling with hardware, 200
SEC, 85
Second processors, 327
SED, 86
SEI, 87
Select 1MHz bus/cartridge, 413
Select input stream, 241
Select output stream, 241
Serial system interrupts, 126
serial port, 240
Serial, 432
Service ROM example, 305
Service ROMs, 294
Set carry flag, 85
Set decimal mode, 86
Set interrupt disable flag, 87
Shadow RAM, 163
shadow memory for display OSBYTE, 158
SHEILA, 152
SHEILA, 403
Sideways scrolling, 201
sideways RAM: allocation, 160
test for OSBYTE, 159
Soft keys, 227
Sound, 372
76489 chip, 375
Speech processor, 379
Speech, 372
STA, 88
Stack Pointer - SP, 29
Stack, 432
Status Register, 28
Store accumulator, 88
Store X, 89
Store Y, 90
string input, 107

STX, 89
STY, 90
STZ, 91
Subtract memory from, 84
System clock, 352
System VIA interrupts, 127
System VIA, 384
TAB character setting, 231
TAX, 92
TAY, 93
Teletext, 205
Test and reset bits, 94
Test and set bits, 95
Test memory bits, 40
Transfer A to X, 92
Transfer A to Y, 93
Transfer S to X, 96
Transfer X to A, 97
Transfer X to S, 98
Transfer Y to A, 99
Transistor transistor logic, 432
TRB, 94
TSB, 95
TSX, 96
TTL, 432
Tube™, 327
TXA, 97
TXS, 98
TYA, 99
ULA, 432
Unknown OSWORD, 111
unknown OSBYTE calls, 111
Unused flag, 29
UPTV, 424
User defined characters, 171
User event, 121
User print vector, 424
User VIA interrupts, 129
User VIA, 381
user flag, 429
user vector (USERV), 425
USERV, 426
USERV: unknown OSWORDs, 111
USR, 12
VDU code table, 166
VDU extension vector, 170
VDU23 expansion, 168
VDUV, 170
Versatile Interface Adapters), 380
vertical sync: event, 120
Video hardware, 187
Video memory use, 184
video system, 165
Word of memory, 432
X Index Register - X, 28
Y Index Register - Y, 28
Z80 OSWORD call, 349
Zero flag - Z, 28
Zero page indirect addressing, 26
Zero page,X addressing, 22
zero page, 14, 20
addressing modes, 22, 26
OS allocations, 114

Programs Disc

A disc of programs from the *New Advanced User Guide* is available from Adder Publishing Limited. The disc contains copies of all the programs from the book to save a considerable amount of typing.

The disc is available in two formats; 5.25in DFS format at 40 tracks (also readable on 80 track drives) for the BBC model B, B+, Master 128 (£12.95 incl. VAT & postage) and 3.5in ADFS format for the Master Compact or Master 128 (£14.95 incl. VAT & postage).

Please send me a copy of the programs disc to accompany the *New Advanced User Guide*. I enclose a cheque/P.O./ or please debit my ACCESS/VISA card for £.....

Name .....

Address .....

.....

City .....Postcode.....

Please state format required:.....

VISA/ACCESS card number:.....

Signature .....

Send to: Adder Publishing Limited,
P.O.Box 148,
Cambridge CB1 2EQ
United Kingdom