


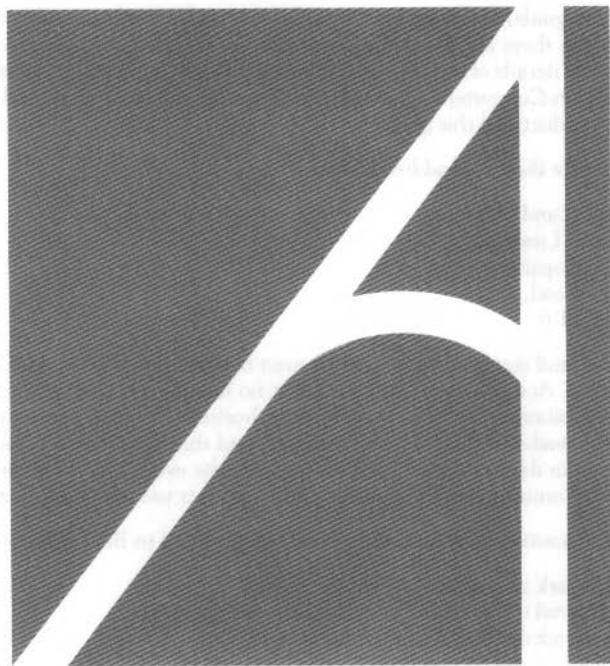
PROGRAMMER'S
REFERENCE
MANUAL



Acorn 

 *rchimedes*

PROGRAMMER'S
REFERENCE
MANUAL



Acorn 

Archimedes

Designed and laser-typeset by Human-Computer Interface Limited, Cambridge.

Copyright © Acorn Computers Limited 1987

Neither the whole nor any part of the information contained in, or the product described in, this guide may be adapted or reproduced in any material form except with the prior written approval of Acorn Computers Limited (Acorn Computers).

The product described in this guide and products for use with it are subject to continuous development and improvement. This applies particularly to the Arthur Machine Operating System, which will be subject to further development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this guide) are given by Acorn Computers Limited in good faith. However, it is acknowledged that there may be errors or omissions in this guide or in the products it describes. A list of details of any amendments or revisions to the guide can be obtained upon request from Acorn Computers. Acorn Computers welcomes comments and suggestions relating to the product and this guide.

All correspondence should be addressed to:

Customer Support and Training,
Acorn Computers Limited,
Cambridge Technopark,
645 Newmarket Road,
Cambridge CB5 8PB.

All maintenance and service on the product must be carried out by Acorn Computers' authorised dealers. Acorn Computers can accept no liability whatsoever for any loss or damage caused by service, maintenance or repair by unauthorised personnel. This guide is intended only to assist the reader in the use of this product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this guide, or any incorrect use of the product.

The British Broadcasting Corporation has been abbreviated to BBC in this guide.

Acorn is a trademark of Acorn Computers Limited.
Econet is a registered trademark of Acorn Computers Limited.
Archimedes is a trademark of Acorn Computers Limited.
Vax is a trademark of the Digital Equipment Corporation.

First published 1987
Issue 1 November 1987
Published by Acorn Computers Limited
Part number 0476,005

ISBN 1 85250 052 2

CONTENTS

INTRODUCTION	1
CONVENTIONS USED IN THIS GUIDE	1
FUNDAMENTAL OPERATING SYSTEM CONCEPTS	3
INTRODUCTION	3
COMMUNICATING WITH THE OPERATING SYSTEM	4
SWI NUMBERS	6
ERROR HANDLING	8
ERROR NUMBERS	9
GENERATING ERRORS	11
OS_BYTE AND OS_WORD	13
SOFTWARE VECTORS	16
HARDWARE VECTORS	30
INTERRUPTS	31
TYPES OF INTERRUPTS	32
EVENTS	36
BUFFERS	42
MISCELLANEOUS OS_BYTES	48
CHARACTER OUTPUT	51
CHARACTER OUTPUT ROUTINES	51
THE OUTPUT STREAMS	54
THE RS423 OUTPUT STREAM	55
THE VDU STREAM	56
THE PRINTER STREAM	57
THE *SPOOL STREAM	63
THE VDU DRIVERS	65
VDU DRIVER CONCEPTS	65
VDU CONTROL SEQUENCES	70
THE VDU OS_BYTES	104
THE VDU OS_WORDS	116
THE VDU SWI CALLS	121
THE MOUSE AND POINTER	128
THE VDU EXTENSION VECTOR	134

CHARACTER INPUT	137
THE INPUT STREAMS	137
BASIC INPUT ROUTINES	139
LINE INPUT	140
THE KEYBOARD	143
THE *EXEC INPUT STREAM	181
THE COMMAND LINE INTERPRETER	183
CALLING THE COMMAND LINE INTERPRETER	183
THE ACTION OF OS_CLI	184
OPERATING SYSTEM COMMANDS	189
FILING SYSTEMS	209
INTRODUCTION	209
FILES, DIRECTORIES AND PATHNAMES	210
FILE TYPES AND DATE STAMPING	213
FILESWITCH AND OS FILING SYSTEM COMMANDS	217
OS FILING SYSTEM SWI CALLS	231
THE ADVANCED DISC FILING SYSTEM	263
THE NETWORK FILING SYSTEM	283
CONFIGURATION COMMANDS	285
INTERFACES	286
NETPRINT, THE NETWORK PRINTING SYSTEM	290
INTERFACES	292
WRITING YOUR OWN FILING SYSTEM	293
MEMORY MANAGEMENT	317
NON-VOLATILE MEMORY (CMOS RAM)	322
HEAP MANAGER	325
MISCELLANEOUS MEMORY SWIs	330
THE PROGRAM ENVIRONMENT	333
RUNNING AN APPLICATION	333
LEAVING AN APPLICATION	334
THE ENVIRONMENT SWIs	335
OPERATING SYSTEM VARIABLES	345
SUMMARY OF EXECUTION MODES	351

MODULES	355
TIME AND DATE	391
NUMBER CONVERSIONS	403
SPRITES	417
THE WINDOW MANAGER	439
THE FONT MANAGER	489
SOUND	519
THE DEBUGGER	567
THE FLOATING POINT EMULATOR	573
ECONET, THE TRANSPORT LAYER	585
PODULE, THE PODULE SYSTEM MANAGER	593
APPENDIX A – ARM ASSEMBLER	595
APPENDIX B – THE LINKER	613
APPENDIX C – ARM PROCEDURE CALL STANDARD	623
APPENDIX D – OPERATING SYSTEM CALLS	641
INDEX	I

INTRODUCTION

This guide describes the Arthur Operating System for the programmer who wants to make the most of the many facilities provided. This operating system is known as the 'Arthur Operating System', but is usually just referred to as the 'OS'. The guide is organised into chapters each of which describes a particular aspect of the system. Because many of the functions are inter-related, you may find that in order to understand a subject fully, you will have to consult several chapters.

At the end of the guide there are some tables which provide a quick reference to many of the operating system calls. You can often locate the information you require by scanning down the appropriate table.

An appendix describes briefly the instruction set of the ARM processor. The operating system call descriptions are oriented towards the assembly language programmer, so some knowledge of ARM assembler is a definite advantage when using this guide.

For information on the ARM chip set (the Acorn Risc Machine, ARM; the memory controller, MEMC; the video controller, VIDC; and the I/O controller, IOC) you are referred to the appropriate **Archimedes Hardware Reference Manual**.

CONVENTIONS USED IN THIS GUIDE

The following conventions are applied throughout this guide:

- Specific keys to press are denoted as `Delete`, `Ctrl`, and so on.
- Text you type on the keyboard and text that is displayed on the screen appears as follows:

```
PRINT "Hello"
```


INTRODUCTION

The operating system (OS) is really just a collection of useful routines. Combined, they provide all the facilities that a programmer might expect to need when writing a complex program. For example, nearly all programs require a way of reading information from the user, and displaying results. The Arthur OS therefore provides a host of routines to perform these important input/output (I/O) operations.

Other aspects of the OS are:

- memory management
- filing system management
- command interpretation
- time and date handling
- data format conversion.

Each of these major topics, and several more, are described in separate chapters.

Some of the OS's work is performed 'behind the scenes'. This refers to the way in which it handles interrupts. An interrupt is a signal to the ARM processor indicating that a device requires attention. The OS will deal with this device by temporarily halting the user's program, and entering what is termed an interrupt routine. This routine deals with the interrupting device very quickly – so quickly, in fact, that you will never realise that your program has been interrupted.

Amongst the devices which are handled under interrupts on the Archimedes are the:

- keyboard
- printer
- RS423 port
- mouse
- disc
- drives
- built-in timers.

Additionally, external hardware may cause new interrupts to be generated. For example, the analogue to digital convertor on the BBC I/O module can interrupt

when it has finished a conversion. It is therefore possible to install routines which deal with these new interrupts.

In fact, much of the OS is concerned with extensibility – allowing you to add to the features already provided to enhance the performance of the machine. A central feature of this is the relocatable module. A module is a piece of software which is loaded into RAM. It conforms to various standards in the way in which it behaves, which means that the facilities provided by the module can be integrated into the system as if they were ‘built-in’. It is very hard to tell whether a facility you are using is provided by the operating system itself, or is really an extension provided by a module.

Because modules are so useful for providing system functions, much of the basic Archimedes software is implemented as modules. Thus, although facilities such as filing systems, sprites, sound, the window manager and fonts are described in this guide as though they were part of the OS, they are really just extension modules which always happen to be present. They differ from other modules, in that they are stored in the machine's ROM, and not loaded into RAM. However they work in the same way. In fact, you can totally replace a ROM module by loading a RAM module of the same name.

COMMUNICATING WITH THE OPERATING SYSTEM

There is only one way in which a program can access an operating system routine, namely by using a SWI instruction. SWI stands for SoftWare Interrupt, and is one of the ARM's built-in instructions. When the ARM executes a SWI, it leaves the current program, and jumps to a fixed location in memory. At this location there is a branch instruction into the operating system ROM. The code in the ROM examines the SWI instruction, and determines which particular OS routine the user wanted. This is called, and when it is finished, control returns to the user's program.

The OS can work out what routine is required because the SWI instruction code contains a 24-bit information field. On the Archimedes, these 24 bits are used to identify a routine uniquely. Thus we talk about SWI numbers. These are the numbers embedded in the SWI instruction. A concrete example is the routine OS_WriteC. This happens to have a SWI number of &00. In assembler it would be written

```
SWI OS_WriteC
```

where OS_WriteC has been set to &00. When this is assembled, the bottom 24 bits of the instruction are set to zero. When the instruction is executed, the OS looks at these 24 bits, and on discovering that they are all zeros, calls the routine to do the OS_WriteC (which prints a character).

In general, you don't have to worry about the exact mechanism used by the OS to decode the SWI instructions. As long as you use the right number after the SWI, the correct result will be obtained. This is particularly easy when you use the BBC BASIC assembler because you can refer to the routines by name. For example, the instruction:

```
SWI "OS_Byte"
```

will be assembled with the correct SWI number for the routine OS_Byte. As long as you spell the name correctly (including the case of the letters), you should never have to worry about SWI numbers.

The prefix of the SWI name (OS in the example above) determines which part of the system will deal with the SWI. OS obviously refers to the calls handled directly by the operating system. Examples of other prefixes are Font, Wimp, ADFS. The prefix is determined by the module which implements the SWI.

Obviously, you need to be able to pass values to OS routines (parameters), and must be able to read values back (results). The ARM registers are used to pass information between the user and the OS. In general, R0 is used to pass the first parameter, and then enough registers after that to pass the rest. It is rare for a routine to use more than 4 or 5 registers.

When the information passed is numeric, character or address, the data itself is generally stored in the register. However, if the data is a string, or a large amount of numeric data, then it passes a pointer to the data instead. For example, filenames are passed as pointers to the characters in memory, and the window manager uses pointers to large window descriptors.

SWI NUMBERS

The 24 bits used to encode the SWI number in the instruction allow SWIs in the range 0 – 16777215 to be used. This SWI 'address range' is divided up into several parts under the OS. For example, SWIs in the range 0 – 262143 provide the basic operating system functions. (Fewer than 150 of these are currently used, however.) Modules can provide their own SWIs, and these must be given unique numbers to avoid clashes.

Finally, you can provide your own SWI actions. When a program executes a SWI whose number is not recognised by the OS or any of the modules in the machine, the OS calls a special routine called the 'Unused SWI vector'. Usually, this will just cause the error `No such SWI`. However, a user program can intercept this and, if the SWI number is one that it recognises, perform the appropriate task.

This section explains in detail how SWI numbers are allocated. The bottom 24-bit section of the SWI op-code is divided up as follows:

Bits 20 – 23

These are used to identify the particular operating system in the machine. All SWIs used under Arthur have these bits set to zero.

Bits 18 – 19

These are used to identify which part of the system software implements the SWI, as follows:

Bit 19	Bit 18	Meaning
0	0	Operating system
0	1	System modules and Acorn extensions
1	0	Third party resident applications
1	1	User applications

Thus OS SWIs, such as OS_WriteC, have both bits clear. Modules such as filing systems and the Font manager have bit 18 of their SWIs set, so their SWI numbers start at &40000. Any software distributed by other software houses should have bit 19 set and bit 18 clear.

Bit 17

This is used to determine the action taken on errors. It is the 'X' bit. Error handling in SWIs is described below.

Bits 6 – 16

These are the SWI Chunk Identification numbers. They identify a block of 64 consecutive SWIs. Anyone wishing to use one of these blocks of SWIs for distributed software should apply in writing to Acorn Computers who will allocate a unique value.

The current Acorn chunk numbers are:

SWI base	Use
&40000	Econet
&40040	NetFS
&40080	Font
&400C0	Wimp
&40100	TUBE
&40140	Sound (level 0)
&40180	Sound (level 1)
&401C0	Sound (level 2)
&40200	NetPrint

&40240	ADFS
&40280	Podule
&402C0	ARMPC
&40300	WaveSynth
&40340	IntelligentInterfaceIEEE
&40380	Debugger
&403C0	SCSIDriver
&40400	VFS
&40440	VideoCommand

Bits 0 – 5

These identify individual SWIs in a chunk. Hence a third party may use SWIs in the following binary range:

000010nnnnnnnnnnnn000000 to 000010nnnnnnnnnnnn111111

where nnnnnnnnnnn is the chunk number that the software house has been allocated.

ERROR HANDLING

It is reasonable to expect that most SWIs can generate an error. For example, if you pass an invalid memory address as a parameter, or a number outside of the expected range, you would expect the SWI routine to tell you about it.

SWIs report errors in a consistent way. If no error occurred, and the desired action was performed, the SWI will clear the ARM's V (overflow) flag on exit. If the SWI routine wishes to indicate that an error did occur, V will be set on exit. Furthermore, R0 will contain a pointer to an error block, which is described below.

Now, just before the operating system passes control back to the user, it checks the V flag. If it is clear (no error) control passes directly back to the user.

If V is set (ie the SWI routine gave an error), the OS performs an action dependent on exactly how the SWI was called in the first place. In particular, if the SWI

number had its bit 17 set, the OS just returns to the user. The V flag will still be set to indicate an error, and R0 will contain the error pointer.

If the SWI number had bit 17 clear, then on detecting the error the OS calls a special error handler routine. This routine is usually set up by the current application, so that when an error occurs, control passes to the application, which deals with the error appropriately. In fact, the OS first notifies modules of the error using a service call, then calls the error vector. (Vectors are a very important part of the operating system, and are described later in this chapter.)

These two types of SWI are known as error-generating (bit 17 clear) and error-returning SWIs. For every SWI, you can call either version, depending on whether you want to detect the error yourself, or leave the current error handler to deal with it.

SWI names also come in two varieties. The error-generating SWIs are of the form we have already seen: `OS_Byte`, `Font_Paint`, etc. Error-returning SWIs, with bit 17 set, are prefixed by an X, eg `XOS_Byte`, `XFont_Paint`.

The error block pointed to by R0 has the following format: the first four bytes form a word containing the error number; following this word is the textual error message, terminated by a zero byte. An error block must be word-aligned, and must be less than 256 bytes long.

ERROR NUMBERS

Just as the 24-bit SWI number is divided into different fields, 32-bit error numbers are also split up. Error numbers are partitioned as described below.

The top byte contains flags:

- Bit 31, if set, implies that the error was a serious one, usually a hardware exception (eg the program tried to access non-existent memory), from which it wasn't possible to return properly with V set.
- Bit 30 is defined to be clear, and can therefore be used by programmers to flag internal errors.

- Bits 24 - 29 are reserved.

The bottom byte is the basic 'error number'.

- The middle two bytes constitute a 'generator' identifier. External authors producing their own errors should apply to Acorn for an identifier.

The following error ranges been reserved:

Range	Error generator
&000 - &0FF	Operating system - BBC-compatible error
&100 - &11F	OS_Module errors
&120 - &13F	OS_ReadVarVal/SetVarVal errors
&140 - &15F	Redirection manager errors
&160 - &17F	OS_EvaluateExpression errors
&180 - &1AF	OS_Claim/Release errors
&1B0 - &1BF	OS_ChangeDynamicArea errors
&1C0 - &1DF	OS_ChangeEnvironment errors
&1E0 - &1EF	OS_CLI/miscellaneous errors
&200 - &27F	Font manager errors
&280 - &2BF	Wimp errors
&2C0 - &2FF	Date/time conversion errors
&300 - &3FF	Econet errors
&400 - &4FF	FileSwitchErrors
&500 - &5BF	Podule errors
&5C0 - &5FF	Printer driver errors
&600 - &63F	General OS errors
&640 - &6FF	International module errors
&700 - &7FF	Sprite errors
&800 - &8FF	Debugger errors
&1XX00 - &1XXFF	Errors from FS number &XX
&10800 - &108FF	ADFS errors

&20000 – &200FF	Sound errors
&20100 – &201FF	SCSI errors
&20200 – &202FF	Video command errors

GENERATING ERRORS

In addition to detecting errors, a program might want to generate an error, to call the current error handler so that the user can find out about the problem. A common example occurs when a program detects that `[Escape]` is pressed. This is usually a sure sign that the user wants to abandon the current operation. The standard response is for the program to acknowledge the escape (see the chapter **CHARACTER INPUT** for details), and generate an `Escape` error. This is then dealt with by the current error handler.

To generate the error, the program calls the SWI `OS_GenerateError`. On entry, `R0` contains a standard error block pointer. The routine never returns to the caller, as it is very difficult to resume after an error. For example, BASIC's error handler will cause the current BASIC program to terminate, returning control to the command mode, or to execute an `ON ERROR` statement, if one is active. We will use `OS_GenerateError` to illustrate the way in which SWIs are documented in this guide. The entry and exit conditions are listed, followed by a short description of the call.

Any registers not mentioned on entry are irrelevant to the call. Any registers not mentioned on exit can be assumed to be preserved.

`OS_GenerateError` &2B (43)

On entry: `R0` = pointer to error block

On exit: either doesn't return or returns with `V` set

`OS_GenerateError` generates an error and invokes the error handler. Whether or not it returns depends on the type of SWI being used. If `XOS_GenerateError` is used, the only effect is to set the `V` flag. This is not very useful.

Here is an example of how OS_GenerateError would be used:

```
SWI "OS_ReadEscapeState"           ;Sets C if escape
BCC noEscape
ADR R0,escapeBlock                 ;Get ptr. to error block
SWI "OS_GenerateError"             ;Do the error - doesn't return
.noEscape
    . . . .
.escapeBlock
    EQU      17                     ;Error number for escape
    EQU      "Escape"+CHR$0        ;Error string
    ALIGN
```

The command line interpreter

One particular SWI is very important because it allows commands to be passed to the OS textually, instead of through machine code instructions. The call OS_CLI takes a pointer to a string. It then interprets that string by executing the command contained within it. There are several commands built into the Archimedes OS, and modules add many more.

The user generally calls the command line interpreter (CLI) by prefixing a command to the current application with a *. The application recognises the * prefix and passes the input line to OS_CLI, instead of trying to execute it itself. The user is said to issue (or execute) a * command.

Because the CLI is so important, it is given a chapter of its own. In that chapter, most of the built-in OS commands are described. The rest of the available commands are described in the chapters appropriate to the command. For example, filing system commands are described in the chapter **FILING SYSTEMS**.

Often, a * command does little more work than call a couple of SWIs. However, whereas it would be very difficult to arrange for a user to be able to call SWIs directly from most applications, it is very easy to allow him or her to type in * commands. Consider the command *TIME. This prints the time and date on the screen. All the command really does is call three SWIs. You can achieve the same effect with a few lines of BASIC:

```
DIM time 5, str 100
?time = 3
SYS "OS_Word",14,time
SYS "OS_ConvertStandardDateAndTime",time,str,100
SYS "OS_Write0",str : PRINT
```

Most people would agree that the * command version is somewhat more convenient.

One disadvantage of using the CLI to perform a task is that it is harder to pass information back to the user. For example, many of the SWIs described in this guide can be used either to set up a system function, or to read the current status of that function without altering its present state. On the other hand, * commands can generally only perform a set or a read operation, not both.

Just as a module may add new SWI numbers to the system, it may also add new * commands. Most of the work of decoding (ie looking up) the command is performed by the OS. The module is called at an address given in its table of commands, and the OS sets up some information for it, such as a pointer to the rest of the command line (so the routine can read any parameters), the number of parameters on the line, and a pointer to the module's workspace.

Finally, like many of the common SWIs, the OS_CLI routine is vectored. This means that its operation may be altered, or replaced entirely, by a routine provided by the user. The section **Software vectors** describes vectors in detail.

OS_BYTE AND OS_WORD

Most SWIs deal with only one task. For example, OS_Module deals with modules, OS_RemoveCursors just removes cursors, and so on. However, there are two calls which perform a wide variety of operations. They are called OS_Byte and OS_Word. They exist, principally, to ease the conversion of software from the BBC and Master series of computers. The operating systems on these machines have two corresponding routines called OSBYTE and OSWORD.

Because the calls are multi-purpose, they tend to appear in more than one chapter of this guide. This section documents the calls in general terms, so that when examples

of their use are given later on, you will understand the entry and exit conditions better.

OS_Byte takes one, two or three parameters. The first parameter, passed in R0, is the reason code. This indicates which particular OS_Byte is required. It has the range 0 – &FF. Thus when we talk about 'OS_Byte &81', this is shorthand for 'OS_Byte with R0 set to &81 on entry'. A complete list of the OS_Byte numbers may be found in the appendix SUMMARY OF OPERATING SYSTEM CALLS.

The second and third parameters are passed in R1 and R2. These too are in the range 0 – &FF; the name OS_Byte comes from the fact that it deals with byte-wide parameters.

The calls may be grouped into three main classes, according to the value of R0 on entry. If R0 < &80 (128), then generally only R1 is used to pass further information. These calls are used for setting, for example, the auto-repeat rate of the keyboard, where only one single-byte quantity is required.

In addition to setting the appropriate system variable, these calls may also perform some other task. For example, OS_Byte &05 sets the printer type, and also waits for the current printer buffer to become empty before returning. Although these calls sometimes return the 'previous' state of whatever is being changed, they are normally used for the action they perform, rather than the information they return.

When R0 is between &80 (128) and &A5 (165), both R1 and R2 are used to hold parameters, and both registers may contain information on exit from the call. The calls are often used for the results they return, rather than to perform particular actions.

For calls with R0 between &A6 (166) and &FF (255) on entry, the action is the same for each value. R0 acts as an index into the RAM which contains certain system variables. These variables are held in consecutive memory locations, with R0=&A6 accessing the first one, R0=&A7 affecting the second one, and so on. The contents of R1 and R2 determine what happens to this system variable, as follows:

New Value = (Old Value AND R2) EOR R1

The most useful application of this rule occurs when the old value is returned without being altered (allowing the status to be read 'non-destructively') as shown below:

R2 = &FF and R1 = &00

and where the value is set to a particular number:

R2 = &00 and R1 = new value

These are the only cases which are stated in the descriptions of OS_Bytes in this guide. Other values of R1 and R2 may be used to alter only selected bits of the variable. For example, to set bits 2 – 4 to the binary pattern 101, and leave the rest unaltered, you would use:

R2 = &E3 (2_11100011) and R1 = &14 (2_00010100)

In all cases, the calls in the range &A6 – &FF return with the previous value of the variable in R1 and the value of the next variable (ie the one which would be accessed with R0+1) in R2.

Many of the calls in this last group access the same system variables as the low-numbered calls, between &00 and &7F. However, as noted above, the lower group may also perform some other action in addition to changing the variable value. This means that the lower group should be used to alter a variable, whereas the upper group may be used for reading the current value without changing it.

Because OS_Bytes perform many useful functions, a * command is provided to call the routine directly. It has the syntax:

```
*FX a
*FX a [, ] b
*FX a [, ] b [, ] c
```

The command is followed by one, two or three parameters, which may be separated by spaces or commas. The values a, b and c are loaded into register R0, R1 and R2 respectively before the OS_Byte is called. Any omitted values are set to zero.

The OS_Word call also takes a reason code in R0. The parameter in R1, which must always be given, is a pointer to a parameter block. This is an area of memory where input parameters are passed to the OS_Word, and where results may be stored. OS_Word is generally used where the two bytes allowed by OS_Byte are not sufficient to pass the required parameters or results. The size of the parameter block varies from call to call, and is documented with each OS_Word description.

Like OS_Byte, OS_Word is multi-purpose, and covers such areas as reading the time and date, setting the screen's 'palette', and reading the definition of a re-definable character. There are far fewer OS_Words than OS_Byte, 0 – &16 being the current range of R0 on entry. This is because calls which require multiple parameters are often given their own SWI numbers, with registers being used to hold the values, as these are more efficient in execution than OS_Words.

SOFTWARE VECTORS

Vectors have already been mentioned a couple of times in this chapter. A vector provides a way of enabling you to change the actions of certain fundamental system routines. When the OS calls such a routine, it does not pass control directly to the code in ROM. Instead, it jumps to the routine pointed to by a memory word – this word is the vector.

Consider the OS_WriteC routine as a concrete example. When the OS decodes a SWI with SWI number &00, it knows that you are requesting a write character operation. Now it could just jump to the appropriate place in the OS ROM. However, this would never allow the action of OS_WriteC to be altered. So instead, the OS loads an address from a memory word – the vector called WriteCV, and passes control to that routine.

Now by default, the WriteCV contains the address of the standard write character routine in ROM. However, you can change this to point to your own routine by calling the SWI OS_Claim, documented later in this section. After you have claimed the vector, whenever an OS_WriteC is executed, your routine will be called, with the same entry conditions. Once your routine has finished, it can either pass the call on, so that the OS write character is still executed, or 'intercept' the call, which will cause control to return to you immediately.

In fact, there may be more than one routine on a given vector. The routines are called in the reverse order to the order in which they called OS_Claim: the last routine to OS_Claim the vector will be the first one called. If that routine passes the call on, the next most recent claimant will get the call, and so on. If any of the routines on the vector intercept the call, the earlier claimants will not be called.

There are some vectors which should not be intercepted; they must always be passed on to other claimants. This is because the default owner, ie the routine which is called if no one has claimed the vector, might perform some important action. The error vector, ErrorV, is a good example. The default owner of this vector is a routine which calls the error handler. If you intercept ErrorV, the error handler will never be called, and errors won't be dealt with properly.

A routine, using a vector, should obey the rules determining which registers are preserved for that call, etc. If passing the call on, the routine should leave all registers intact, unless they are deliberately altered to change the effect of the call. If intercepting the call, the routine should preserve all unused registers to pass back results.

Vector SWI calls

The SWIs which control the use of vectors are documented below:

OS_Claim &1F (31)

On entry: R0 = vector number
R1 = address of claiming routine
R2 = value to be passed in R12 when the routine is called

On exit: -

This call adds the routine whose address is given in R1 to the list of routines intercepting the vector. This becomes the first routine to be used when the vector is called.

The R2 value enables the routine to have a workspace pointer set up in R12 when it is called. If the routine using the vector is in a module (as will often be the case), this pointer will usually be the same as its module workspace pointer.

See below for a list of the vector numbers.

Example: `MOV R0, #ByteV`
`ADR R1, MyByteHandler`
`MOV R2, #0`
`SWI "OS_Claim"`

OS_Release &20 (32)

On entry: R0 = vector number
R1 = address of releasing routine
R2 = value given in R2 when claimed

On exit: -

This removes the routine, which is identified by both its address and workspace pointer, from the list for the specified vector. The routine will no longer be called.

Example: `MOV R0, #ByteV`
`ADR R1, MyByteHandler`
`MOV R2, #0`
`SWI "OS_Release"`

OS_CallAVector &34 (52)

On entry: R0 – R8 = vector routine parameters
R9 = vector number

OS_CallAVector calls the vector number given in R9. R0 – R8 are parameters to the vectored routine; see the descriptions below for details. This is used for calling vectored routines which don't have any other entry point, such as the InsV vector.

OS_UpCall &33 (51)

On entry: R0 = reason code
R1 - R3 depend on reason code

On exit: R0 depends on response to upcall

OS_UpCall calls the UpCall vector. This is called in (potentially recoverable) error circumstances by the system, so corrective action can be taken. For example, ADFS executes an OS_UpCall before issuing a Disc not present error, so the application in control of the screen can print an appropriate prompt for the disc to be inserted.

There are two currently defined reason codes, as follows:

R0 = 1

This is issued by the filing system to indicate 'medium not present'. If the code on the UpCall vector can take corrective action, such as by prompting the user to insert the correct disc, then the code should return (by intercepting the vectored call) with R0 = 0. If no action can be taken, the call should be passed on.

When the UpCall vector is called, the following registers are set up:

R1	filing system number, eg 8 for ADFS
R2	pointer to a null-terminated disc name required, or -1
R3	device number, if appropriate, or -1 if not

R0 = &100

This call is made just before a new application is going to be started (eg due to a *RUN or module command). If for some reason you don't want the application to start, you should set R0 to 0 and claim the call. This will cause the error Unable to start application to be given. Otherwise, you should pass the call on with all registers intact.

Writing vector code

A routine using a vector has to obey certain rules. It must obey the entry and exit conditions of the OS routine in question. For example, a routine on WriteCV must preserve all registers. Routines intercepting a vector are allowed to return errors by setting the V flag, and storing the error pointer in R0.

The processor mode in which the routine is entered depends on the vector. Most vectors are called in SVC mode. This is the mode used when the SWI instruction is executed. An important thing to remember about this mode is that if you call another SWI, R14 will be corrupted. This is not the case when a user-mode program calls a SWI. Consequently, the routine should use the (full, descending) stack addressed by R13 to save R14 first.

The event vector is called in IRQ mode, and has access to the IRQ stack. From within the event code, all the usual restrictions of interrupt routines must be observed.

One of two methods is used to return from a vectored routine. If a routine wishes to pass on the call (to the previous owner), it should return by copying the value of R14 with which it was entered into the PC. This can be done by:

```
MOV PC,R14
```

If you wish to intercept the call, it should pull an exit address (which has been set up by the operating system) from the stack and jump to it. This should be done by:

```
LDMFD R13!, {PC}
```

Sometimes, a routine may want to call the previous owner of the routine, and modify the results that it produced before returning to the original caller. This may be achieved as follows:

```

STMFD R13!, {R9}           ;Save work register
ADR   R9, cont+SVC_Mode    ;'Fudge' a return address on the
STMFD R13!, R9             ;stack for previous owner
MOV   PC, R14              ;Call previous owner
.cont                               ;the previous owner returns here
....                               ;Process results returned
....
LDMFD R13!, {R9,PC}       ;Restore R9 and claim call

```

The return address, pushed on the stack for the default owner, must also have the correct mode bits set, in this example SVC_Mode (the bottom two bits of R15 set).

Vector descriptions

The operating system vectors are:

ErrorV	(&01)	Error vector (OS_GenerateError)
IrqV	(&02)	Interrupt vector
WriteCV	(&03)	Write character vector (OS_WriteC)
ReadCV	(&04)	Read character vector (OS_ReadC)
CliV	(&05)	Command line interpreter vector (OS_CLI)
ByteV	(&06)	OS_Byte indirection vector
WordV	(&07)	OS_Word indirection vector
FileV	(&08)	File read/write vector (OS_File)
ArgsV	(&09)	File arguments read/write vector (OS_Args)
BGetV	(&0A)	File byte read vector (OS_BGet)
BPutV	(&0B)	File byte put vector (OS_BPut)
GBPBV	(&0C)	File byte block get/put vector (OS_GBPB)
FindV	(&0D)	File open vector (OS_Find)
ReadLineV	(&0E)	Read a line of text vector (OS_ReadLine)
FSControlV	(&0F)	Filing system control vector (OS_FSControl)
EventV	(&10)	Event vector (OS_GenerateEvent)
InsV	(&14)	Buffer insert vector
RemV	(&15)	Buffer remove vector
CnpV	(&16)	Count/Purge Buffer Vector
UKVDU23V	(&17)	Unknown VDU23 vector (OS_WriteC)
UKSWIV	(&18)	Unknown SWI vector (SWI)

UKSWIV	(&18)	Unknown SWI vector (SWI)
UKPLOTV	(&19)	Unknown VDU25 vector (OS_WriteC)
MouseV	(&1A)	Mouse vector (OS_Mouse)
VDUXV	(&1B)	VDU vector (OS_Write)
TickerV	(&1C)	100Hz pacemaker vector
UpcallV	(&1D)	Warning vector (OS_UpCall)
ChangeEnvironmentV	(&1E)	Environment change warning (OS_ChangeEnvironment)

The names of the routines which can cause the vector to be called are in brackets.

More details of these vectors are given below. For the vectors which correspond to OS calls documented elsewhere, you should see those sections for the entry and exit conditions of the vector.

Note that the filing system vectors FileV (&08) to OpenV (&0D) have 'no default action', ie they return immediately. However, the FileSwitch module (described in the chapter **FILING SYSTEMS**) OS_Claims the vectors whenever the machine is reset, so effectively the default action is to perform the appropriate filing system routine.

ErrorV (&01) – Error vector

On entry: R0 = pointer to an error block

On exit: –

This vector is called when an error is generated. When the end of the chain is reached (ie when the default vector owner is called), the error handler is called. The error block pointed to by R0 consists of a one-word error number followed by a null-terminated error string.

- *Note:* it is important that routines on this vector do not intercept it. If they do, the owner of the error handler will never be called, and strange things will occur. You should view this as a 'warning' vector, which gives you the opportunity, for example, to set an internal flag before passing it on.

IrqV (&02) – Unknown interrupt vector

On entry: Processor in IRQ mode, IRQs disabled

On exit: –

This vector is called when an unknown IRQ is detected. No information is passed to the routine. The routine finds out if the device causing the interrupt request is its responsibility, and if so deals with it appropriately. If this is done, the call should be claimed, otherwise (if your routine does not recognise the interrupt) it is passed on.

The routine preserves all registers, keeps IRQs disabled, and returns as quickly as possible. The default action is to generate an Unknown IRQ error.

WrchV (&03) – Write character vector

This vector is used to indirect all calls to OS_WriteC. The default action is to call the ROM write character routine.

RdchV (&04) – Read character vector

This vector is used to indirect all calls to OS_ReadC. The default action is to call the ROM read character routine.

ClivV (&05) – Command line interpreter vector

This vector is used to indirect all calls to OS_CLI. The default action is to call the ROM command line interpreter.

ByteV (&06) – OS_Byte indirection vector

This vector is used to indirect all calls to OS_Byte. The default action is to call the ROM OS_Byte routine.

WordV (&07) – OS_Word indirection vector

This vector is used to indirect all calls to OS_Word. The default action is to call the ROM OS_Word routine.

FileV (&08) – File read/write vector

This vector is used to indirect calls to OS_File. See the note above for the default action.

ArgsV (&09) – File arguments read/write vector

This vector is used to indirect calls to OS_Args. See the note above for the default action.

BGetV (&0A) – File byte read vector

This vector is used to indirect calls to OS_BGet. See the note above for the default action.

BPutV (&0B) – File byte put vector

This vector is used to indirect calls to OS_BPut. See the note above for the default action.

GBPBV (&0C) – File byte block get/put vector

This vector is used to indirect calls to OS_GBPB. See the note above for the default action.

OpenV (&0D) – File open vector

This vector is used to indirect calls to OS_Find. See the note above for the default action.

ReadlineV (&0E) – Read line vector

This vector is used to indirect calls for SWI OS_ReadLine. The default action is to call the ROM OS_ReadLine routine.

FSCV (&0F) – Filing system control vector

This vector is used to indirect calls to OS_FSControl. See the note above for the default action.

EventV (&10) – Event vector

This vector is called by OS_GenerateEvent (which all event-generating routines use). For details on the entry conditions see the section **Events** below. The default action is to call the event handler. See the chapter **THE PROGRAM ENVIRONMENT** for details of this and other handlers.

InsV (&14) – Buffer insert vector

On entry: R0 = character to be inserted
R1 = buffer number

On exit: R2 is undefined
C = 1 implies insertion failed

This routine is called by OS_Byte &8A and OS_Byte &99. The default action is to call the ROM routine to insert a character into a buffer. It may also be called using OS_CallAVector. It should be called with interrupts disabled (the OS_Bytes do this automatically).

See the section **Buffers** for a description of the available buffers.

RemV (&15) – Buffer remove vector

On entry: R1 = buffer number
R3 = 1 if buffer to be examined only, else R3 = 0

On exit: R0 = next character to be removed (for examine option)
R2 = character removed (for remove option)
C = 1 means buffer was empty on entry

This vector points to the operating system routine used to remove a character from a buffer or to examine the next character to be removed. It is used by OS_Byte &91 and OS_Byte &98. The default action is to call the ROM routine to inspect or remove a character from a buffer.

If the remove option is used then the character is returned in R2. If the buffer was empty then the carry flag is returned set.

CnpV (&16) – Count/purge buffer vector

On entry: R1 = buffer number
R3 = desired effect
C = return value flag, if R3=0

On exit: R0 is undefined
R1 is count, if R3=0 on entry

This vector points to the operating system routine used to count the number of entries in a buffer or to purge the contents of a buffer. R3 determines the operation as follows:

R3 = 0 count the entries in a buffer
R3 = 1 purge the buffer

If the entries are to be counted then the result returned depends on the carry flag on entry as follows:

C = 0 return the number of entries in the buffer
C = 1 return the amount of space left in the buffer

This vector is used by OS_Bytes &0F, &15 and &80.

UKVDU23V (&17) – Unknown VDU23 vector

On entry: R0 = VDU 23 option requested
R1 = pointer to VDU queue

On exit: –

This vector points to a routine to handle the case when VDU 23,n has been issued, and n is not a recognised value, ie it is in the range 18 – 25 or 28 – 31. The nine parameters sent after the VDU 23 command are stored in the VDU queue. R1 points to the byte holding n, and R0 also contains n.

The default action is to do nothing, ie unknown VDU 23s are usually ignored.

SWIV (&18) – Unknown SWI vector

On entry: R0 – R9 as set up by the caller
R11 = SWI number

On exit: –

This vector points to a routine to handle the case when a SWI call has been issued with an unknown SWI value. Before this vector is called, the OS tries to pass the call to any modules which have SWI table entries in their header.

The default action is to call the unknown SWI handler. The usual action of this handler is to give a **No such SWI** error. See the chapter **THE PROGRAM ENVIRONMENT** for more details of how the SWI handler is called.

UkPlotV (&19) – Unknown VDU 25 vector

On entry: R0 = PLOT number

On exit: –

This vector points to a routine to handle the case when an unrecognised VDU 25 (PLOT) number has been used. The co-ordinates of the last three points, which

have been visited, may be read using OS_ReadVduVariables. The contents of VDU variables 138 – 147, at the point when the vector is called, are:

GCsX	Co-ordinates given in unknown plot command (external)
OlderCsX	Last but two point visited (internal)
OldCsX	Last but one point visited (internal)
GCsIx	Last point visited (graphics cursor) (internal)
NewPtX	Co-ordinates given in unknown plot command (internal)

When the call returns, the VDU drivers update the variables, so that the point given in the unknown plot becomes the graphics cursor position. The previous graphics cursor becomes the last point but one, the previous last point but one becomes the last point but two, and the previous last point but two is lost.

The default action is to do nothing, ie unknown plot commands have no effect.

MouseV (&1A) – Mouse vector

On entry: –

On exit: R0 = x position of mouse
R1 = y position of mouse
R2 = button state
R3 = monotonic time of the reading

This is the vector called by OS_Mouse. The default action is to examine the mouse buffer. If there is an entry, this is used to form the return registers. If the buffer is empty, the current state is read, and the current monotonic time is used for R3.

VDUXV (&1B) – VDU extension vector (under certain conditions only)

On entry: R0 = Byte sent to the VDU

On exit: –

This vector points to a routine to handle calls to the VDU when bit 5 of the OS_WriteC destination flag is set. When this bit is set, all characters sent to the

VDU driver are routed through this vector instead. Note that this only affects the display driver: other output streams such as the printer and *SPOOL file are called as usual, even when VDUXV is used for screen updating.

It is up to the owner of the vector to perform the usual queuing of parameter bytes etc. The default owner of this vector does nothing, so issuing a *FX3,32 call is much the same as disabling the VDU using ASCII 21.

The font manager uses this vector. See the chapter **VDU DRIVERS** for more details on the VDU streams.

TickerV (&1C) – 100Hz pacemaker vector

On entry: processor in IRQ mode

On exit: –

This vector is called every centi-second. It should never be intercepted, as this might prevent other users from being called. A typical use of this call is in writing an interrupt driven custom printer driver.

UpCallV (&1D) – Warning vector

On entry: See SWI OS_UpCall above

On exit: See SWI OS_Upcall above

This vector is called by OS_UpCall. The entry and exit conditions for that call are documented above. The default action is to return immediately.

ChangeEnvironmentV (&1E) – Warning of change in environment

On entry: R0 = change handle
R1 = new value
R2 = R12 to be called with
R3 depends on R0

On exit: R1 = 0 to stop change from taking place

This vector is called whenever a program issues an OS_ChangeEnvironment. It is called with the same entry conditions as that routine. If you have OS_Claimed this vector, and if you do not want the change to occur, you should set R1 to 0 and pass the call on. See the above-mentioned SWI for more details of when this is used (and the chapter **THE PROGRAM ENVIRONMENT**).

HARDWARE VECTORS

The hardware vectors are a set of words starting at logical address &0000000. They are used by the ARM processor in certain exceptional circumstances. Usually, a vector will contain a branch to a routine to handle the exception. The vectors and their addresses are:

Address	Vector	Default contents
&00	Reset	B branchThru0Error
&04	Undefined instruction	LDR PC, UndHandler
&08	SWI	B decodeSWI
&0C	Prefetch abort	LDR PC, PabHandler
&10	Data abort	LDR PC, DabHandler
&14	Address exception	LDR PC, AexHandler
&18	IRQ	B handleIrq
&1C	FIQ	FIQ code

The Reset vector is always under control of the hardware; it can never be usefully altered because on resetting the machine, ROM is temporarily switched into location zero, so the Reset vector will always be the same. Any attempt to jump to this location will result in a Branch through zero error.

The middle group of vectors, except SWI, are under the control of various 'environment' handlers. These may be set and read as described in the chapter **THE PROGRAM ENVIRONMENT**. Very few programs need to take account of these vectors. The usual action of these exceptions is to cause an error. If the floating point

emulator is active, it intercepts the undefined instruction vector to interpret floating point instructions.

The SWI vector contains a branch to the OS code which determines the SWI number and branches to the appropriate location. If no built-in or module SWI corresponds to the number given, the unused SWI vector is called. It is not recommended that you replace this vector.

The IRQ vector also contains a branch into the OS code. This code attempts to deal with the interrupt by examining the hardware devices that are expected to interrupt the processor. If none of these prove to be the source of the interrupt, the software vector IrqV is called.

Finally, FIQ doesn't usually contain a branch, but is the first instruction of a RAM-based routine to deal with the FIQ. See the next section for more details on IRQ and FIQ.

INTERRUPTS

Interrupts are signals from various hardware devices to the CPU to tell the processor that a device requires attention. They are sent, for example, when a key has been pressed or when one of the software timers needs updating. They provide a very efficient means of control since the processor doesn't have to be responsible for regularly checking to see if it needs to deal with any background tasks of this nature. Instead, it can concentrate on executing your code or whatever else its current main task may be, and only deal with the background tasks when necessary.

If a device wants attention, it alters the status of its interrupt request pin, setting it either high or low depending on the particular device. Then, if interrupts are enabled, the CPU:

- finishes executing the current instruction
- stores the program counter in the interrupt mode's R14
- disables interrupts so that the interrupt routine may not be interrupted

- then passes through the hardware interrupt vector to the interrupt handling routine.

The routine must discover which device requested the interrupt. Any device, which is capable of requesting interrupts, has a status byte mapped in memory, of which one particular bit is set if it is requesting an interrupt. Hence, the interrupt bits of all these registers are checked in order from the most time-sensitive to the least time-sensitive to find out which device put in the request. The interrupt condition must be cleared to prevent further interrupts. The method of doing this depends on the device. However, a common method is for the status byte to be 'read sensitive' so that the act of reading the value in it automatically clears the interrupt request bit.

Once the interrupt routine for the device has been executed a 'return from interrupt' is performed. This is done using the instruction:

```
SUBS PC, R14, #4
```

The R14 used is the IRQ/FIQ mode one. The CPU then continues executing the interrupted program at the next instruction.

You are not likely to have to employ such code yourself, as interrupts, like all other aspects of the machine's operation, are under the control of the OS.

TYPES OF INTERRUPTS

There are actually two classes of interrupts, normal interrupts and fast interrupts. Fast interrupts are generated by devices which demand that their request is dealt with as quickly as possible. These fast interrupts have a separate interrupt request pin from the normal interrupts and are passed through a different vector; see the previous section.

Intercepting interrupts

A program wishing to use IRQ-type interrupts should OS_Claim the software vector IrqV. Once the OS has determined that it doesn't know about the interrupting device, it calls IrqV to deal with it. This can lead to quite slow responses if there

happens to be an interrupt from a device that the OS does know about at the same time as your program's hardware, the OS will deal with that first.

It is possible to intercept the IRQ routine before the OS does its checking. The routine to which the hardware IRQ vector at location &18 branches, contains code which is similar to the following:

```

SUB    R14, R14, #4           ;Get actual return address
STMFD  R13!, {R12,R14}       ;Save it and R12 on IRQ stack
MOV    R12, #0
ADR    R14, return           ;Set up return address
LDR    PC, [R12,#&100]       ;Call vector at location &100
.return                               ;Test call back flag

```

So, if you put a pointer to your routine at location &100, this will be called whenever an IRQ occurs. You can then make a quick check for the interrupting device and if it is not one of yours, jump to the previous contents of &100 – having preserved all registers.

If the interrupt was yours, you should handle it, and return using R14. Again, all registers should be preserved, except R12, which may be corrupted.

Note that no workspace pointer is set up if you use this method, so any workspace you use will have to rely on the stack. As you must have a copy of the old contents of location &100 which has to be stored with the code, the routine must be RAM-based.

Here are some general points regarding interrupt (and event) routines:

It is strongly advised that the interrupt routine should not re-enable interrupts. If it does, then your interrupt routine should be able to handle a second interrupt occurring and hence being entered a second time before the first is finished.

The interrupt routine should avoid calling certain operating system routines since it may call the one which the foreground process was using when the interrupt occurred. If this routine is not re-entrant, the foreground process will be adversely affected. In general, OS_Byte and OS_Word calls can be used. Any particular calls

which cannot be used are documented as such. OS_WriteC and routines which use it should never be called.

Before a SWI is called from an interrupt routine, the register R14_SVC must be preserved, and restored once the routine returns. The recommended way of doing this is:

```
MOV      R9, PC           ;Save current status/mode
ORR      R8, R9, #SVC_Mode ;Derive SVC-mode version of it
TEQP     R8, #0          ;Enter SVC mode
MOVNV    R0, R0          ;No-op to prevent contention
STMFD    R13!, {R14}     ;Save R14_SVC
SWI      XXXX            ;Do the SWI
LDMFD    R13!, {R14}     ;Restore R14_SVC
TEQP     R9, #0          ;Re-enter original processor mode
MOVNV    R0, R0          ;No-op
```

SVC_Mode is 3. Of course, you must preserve R8 and R9.

FIQs are intercepted in a different way from IRQs. There is a default owner of the (hardware) FIQ vector. This is the Econet module, if present. If anyone else wants to use FIQs, for example to perform a disc transfer under interrupts, you should claim the FIQ vector for the time required. Having claimed the FIQ vector (using the appropriate service call), and ensured that no FIQs are generated immediately, the claimer should poke the new FIQ-handling routine into addresses &1C onwards. Up to location &100 is available (ie the last possible instruction is at &FC).

When the FIQ operation is complete, the claimer should release the FIQ vector again. Claiming and releasing the FIQ vector is performed using a module service call. See the chapter **MODULES** for details. In summary, the sequence of events is:

- Claim FIQs
- Set-up FIQ code at &1C...
- Enable your FIQ device, and only the appropriate bit in the IOC FIQ mask

- Perform FIQ operation
- Disable your FIQ device having set the IOC FIQ mask to zero
- Release FIQs.

The rules for writing FIQ code are much the same as IRQ code. It is even more important that the code returns quickly.

Disabling interrupts

Although both types of interrupt are maskable, these masks may not be set when the processor is in user mode. Programs generally execute in user mode, so you must use special techniques if you want to disable interrupts.

It is possible to disable interrupts (IRQs only) by calling `OS_IntOff` (no entry or exit conditions). However, this should be used with care, and particularly not for long periods of time since this will have various unwanted effects such as stopping the clock, disabling the keyboard, etc. IRQs may be re-enabled again by calling `OS_IntOn`.

Alternatively, you may enter SVC mode using `OS_EnterOS`. When the processor is in this mode, the IRQ and FIQ masks may be changed at will. It is not recommended that you execute programs in SVC mode, as you will be using a small stack, and R14 will have to be preserved between calls to SWIs etc. To return to user mode after an `OS_EnterOS`, use a `TEQP` instruction. For example, the code below disables both types of interrupts and returns to user mode:

```
SWI    "OS_EnterOS"           ;Enter SVC mode
MOV    R0,PC                  ;Get status in R0
ORR    R0,#&0C000000          ;Set the interrupt masks
TEQP   R0,#3                  ;Update PSR, and return to user mode
```

EVENTS

The operating system performs a large number of background tasks by servicing interrupts on a regular basis. Whilst carrying out these tasks the operating system may generate one or more of the following events which indicate that a particular situation has occurred:

Number	Event type
0	Output buffer has become empty
1	Input buffer has become full
2	Character has been placed in input buffer
4	Electron beam has reached last displayed line (Vsync)
5	Interval timer has crossed zero
6	ESCAPE condition has been detected
7	RS423 error has been detected
8	Econet has generated an event
9	User has generated an event
10	Mouse buttons have changed state
11	A key has been pressed or released
12	Sound event

To use an event, you must first OS_Claim the event vector, then enable the required event(s). By default, all events are disabled. They may be enabled by using OS_Byte &0E (14) and disabled using OS_Byte &0D (13), as follows:

OS_Byte &0D (13) – Disable event

On entry: R1 = event number

On exit: R1 = old enable state
R2 is undefined

The previous enable state of the event is returned in R1:

R1 = 0 previously disabled
R1 > 0 previously enabled

Note that to disable an event totally, you must use OS_Byte &0D (13) the same number of times as you use OS_Byte &0E (14).

OS_Byte &0E (14) – Enable event

On entry: R1 = event number

On exit: R1 = old enable state
R2 is undefined

This call provides a means of enabling specific events (see above).

The previous enable state of the event is returned in R1:

R1 = 0 previously disabled
R1 > 0 previously enabled

In addition to the OS interrupt routines generating events, user programs may also cause them. An event number, 9, has been reserved for such events. You cause an event using OS_GenerateEvent:

OS_GenerateEvent &22 (34)

On entry: R0 = event number
R1... = event parameters

On exit: –

Note that, as usual, the event vector will only be called if the event number given in R0 has previously been enabled using OS_Byte &0E.

When an event occurs, your event routine is entered with the event number stored in register R0 and possibly other information in R1 onwards, depending on the event.

The notes which apply to interrupt handlers also apply to event handlers – namely, event routines are entered with interrupts disabled, with the processor in a non-user mode. They should not re-enable interrupts, and the use of certain operating system calls should be avoided. See the code fragment in the section **Intercepting interrupts** for the way in which SWIs must be called.

Details of all the events and values passed to the event routines are given below.

Output buffer has become empty (0)

On entry: R1 = buffer number

This event is generated when the last character has just been removed from an output buffer (eg printer buffer, RS423 output buffer). Buffers are discussed in the next section **Buffers**.

Input buffer has become full (1)

On entry: R1 = buffer number
R2 = Buffer value of character

This event is generated when an input buffer is full and when the operating system tries to enter a character into the buffer but fails. See the discussion on buffers below.

Character has been placed in the keyboard input buffer (2)

On entry: R2 = Buffer value of key

This event is generated when a key is pressed, independent of the input stream selected. See the chapter **CHARACTER INPUT** for a description of buffer values for the keyboard buffer.

Vsync – Electron beam has reached last displayed line (4)

On entry: –

This event is generated every fiftieth of a second. It occurs when the electron beam reaches the bottom of the displayed area and is about to start displaying the border colour. This event corresponds to the time when the OS_Byte &19 call returns to the user.

It could be used, for example, to start a timer which will cause a subsequent interrupt. On this interrupt, the screen palette might be changed, to allow more than the usual number of colours on the screen at once.

Interval timer has crossed zero (5)

On entry: –

This event is generated when the interval timer, which is a five-byte value incremented 100 times a second, has reached zero. See OS_Word &03 for details of the interval timer.

Escape condition has been detected (6)

On entry: –

This event is generated when either Escape is pressed or when an escape condition is received from the RS423 input port. See the chapter CHARACTER INPUT for a discussion of escape conditions.

RS423 error has been detected (7)

On entry: R1 = pseudo 6850 status register shifted right 1 place
R2 = character received

This event is generated when an RS423 error is detected. Such errors are parity errors, framing errors etc. On entry, the bits of R1 have the following meanings:

Bit	Meaning when set
-----	------------------

5	Parity error
4	Over-run error
3	Framing error

Econet has generated an event (8)

On entry: -

This event is generated when an Econet event is detected.

User event (9)

On entry: --

This event is generated when the user calls OS_GenerateEvent with R0=9. The other registers are as set up by the user. Note that this is entered in SVC mode, not IRQ mode.

Mouse button event (10)

On entry: R1 = mouse X co-ordinate
R2 = mouse Y co-ordinate
R3 = button state
R4 = 4 bytes of monotonic centi-second value

This event is generated when a mouse button changes, ie when a button is pressed or released. The button state is given in R3 as follows:

Bit	Meaning when set
0	Right-hand button down
1	Centre button down
2	Left-hand button down

Key up/down event (11)

On entry: R1 = 0 for key up, 1 for key down
R2 = key number
R3 = keyboard driver ID

This event is issued whenever a key on the keyboard is pressed or released. The key number, R2, is an low-level internal key number, which does not relate to other codes used elsewhere. The table below lists the values for each possible key, giving the high and low hex digit of the key code:

	low	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
high	0	Esc	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	Pr	SL	Brk
1	'	1	2	3	4	5	6	7	8	9	0	-	=	`	<-	Ins	
2	Hme	PgU	NL	/	*	#	Tab	Q	W	E	R	T	Y	U	I	O	
3	P	[]	\	Del	Cpy	PgD	7	8	9	-	CtL	A	S	D	F	
4	G	H	J	K	L	:	"	Ret	4	5	6	+	ShL		Z	X	
5	C	V	B	N	M	,	.		ShR	Up	1	2	3	CL	ALL	SpC	
6	AlR	CtR	Lft	Dwn	Rt	0	.	Ent									

Where there is some ambiguity, eg the digit keys, it should be clear from referring to the keyboard layout which code refers to which key. The keys are numbered top to bottom, left to right, starting from Escape at the top left corner.

Note that the keycodes given in this event bear little relationship to any other code the user is likely to see. They are not, for example, related to the negative INKEY numbers described in the chapter CHARACTER INPUT.

Sound event (12)

On entry: R1 = sound level number (2)
R2 = 0

Currently, the only sound event is generated by the level 2 scheduling software, so R1=2. The 0 in R2 may change in future versions to give the invocation number of the task causing the event.

The event is generated whenever the sound beat counter is reset to zero, marking the start of a bar. See the chapter **SOUND** for more details.

BUFFERS

On the Archimedes much of the transfer of data is performed under interrupts, therefore there is extensive use of buffers. These act as temporary holding areas for data after it is generated by the user (or device) and before it is consumed by the device (or user). For example, whenever you type a character on the keyboard, that character is stored in the keyboard input buffer by the keyboard interrupt handler, and it remains there until the program is ready to use it.

We are not concerned with filing system buffers in this section. However, these are areas where whole sectors of a disc are held in memory to increase the efficiency of the disc accesses. The use of disc buffers is generally invisible to the user, who has no direct way of accessing their contents.

The buffers under discussion are known as first-in first-out, or FIFO, buffers. This is because the characters are removed from the buffer in the same order in which they were inserted. Many operations on buffers are implicit. For example, when you send a character to the printer or RS423 port, a character is inserted into a buffer. When you read from the keyboard or RS423 port using `OS_ReadC`, a character is removed from the buffer.

Additionally, there are several explicit buffer operations available. These include:

- inserting a character into a buffer
- removing a character
- counting the space in a buffer
- examining the next character without removing it
- flushing a buffer (clearing its contents).

These are implemented as `OS_Bytes`, and are documented below. The flush operation is also performed implicitly when the escape condition is cleared (see the chapter **CHARACTER INPUT**).

Buffer numbers

There are ten buffers, numbered 0 – 9. Their uses are as follows:

Number	Use	Size
0	Keyboard	31
1	RS423 (input)	255
2	RS423 (output)	255
3	Printer	63
4	Sound channel 0	15
5	Sound channel 1	15
6	Sound channel 2	15
7	Sound channel 3	15
8	Speech	63
9	Mouse	63

Buffers 2 to 8 are output buffers. They hold data generated by the user until a device is ready to consume it. The others are input buffers. These store bytes generated by the keyboard, RS423 and mouse respectively until the user is ready to read them.

Currently, buffers 4 to 8 are not used. They are provided for compatibility with BBC Micro software. Sound buffering is performed differently on the Archimedes from the BBC, and speech is also implemented in a different way. These buffers are not considered further.

The format of data in all buffers in current use, except for the mouse buffer, is byte-oriented ASCII data. The mouse buffer contents refer to buffered key clicks. The format is as follows:

Byte	Value
0	Mouse x co-ordinate low
1	Mouse x co-ordinate high
2	Mouse y co-ordinate low
3	Mouse y co-ordinate high
4	Button state
5	Time of button change, byte 0
6	byte 1
7	byte 2
8	Time of button change, byte 3

The bytes are listed in the order in which they would be removed using OS_Byte &91 (145). Usually OS_Mouse reads data from the mouse buffer. If none is available, it returns the current state instead. The mouse buffer is 63 bytes long, so 7 entries may be held at once.

Buffer OS_Byte calls

OS_Bytes for handling buffers are described below.

OS_Byte &0F (15) – Flush buffer

On entry: R1 = action code

On exit: R1 is undefined
R2 is undefined

This call empties either all the buffers or only the current input buffer:

R1 = 0 flush all buffers
R1 = 1 flush the current input buffer (keyboard/RS423)

The contents of the buffer(s) are discarded. Individual buffers may be flushed using OS_Byte &15 (21).

OS_Byte &15 (21) – Flush selected buffer

On entry: R1 = buffer number

On exit: R2 is undefined

OS_Byte &80 (128) – Get buffer/mouse status

On entry: R1 = action code

On exit: R1 = low byte of position or number of bytes in buffer/free
R2 = high byte of position or number of bytes in buffer/free

The action of this call depends upon the value in R1. It determines either the current x or y position of the mouse, or the number of free bytes in a particular input buffer, or how many bytes there are free in a particular output buffer:

On entry On exit

R1 = 7	R1 contains x position MOD 256; R2 contains x position DIV 256
R1 = 8	R1 contains y position MOD 256; R2 contains y position DIV 256
R1 = 255	R1 & R2 contain the number of bytes in the keyboard buffer
R1 = 254	R1 & R2 contain the number of bytes in the RS423 input buffer
R1 = 253	R1 & R2 contain the number of bytes free in the RS423 output buffer
R1 = 252	R1 & R2 contain the number of bytes free in the printer buffer
R1 = 251	R1 & R2 contain the number of bytes free in sound channel 0
R1 = 250	R1 & R2 contain the number of bytes free in sound channel 1
R1 = 249	R1 & R2 contain the number of bytes free in sound channel 2
R1 = 248	R1 & R2 contain the number of bytes free in sound channel 3
R1 = 247	R1 & R2 contain the number of bytes free in the speech buffer
R1 = 246	R1 & R2 contain the number of bytes in the mouse buffer

Obviously we are more concerned with the calls where R1 \geq 247 here. Note that R1 = $-(\text{buffer number} + 1) \text{ AND } \&\text{FF}$ (or $255 - \text{buffer number}$).

On exit, R1 contains the low byte of the count, and R2 contains the high byte. For input buffers the count is the number of bytes in the buffer. For output buffers it is the number of free bytes left in the buffer.

OS_Byte &8A (138) – Insert character code into buffer

On entry: R1 = buffer number
R2 = value

On exit: C = 1 if buffer is full

This call inserts the number specified in R2 into the buffer identified by R1. If C=1 on exit, the character was not inserted as there was no room. Inserting characters into the mouse buffer isn't recommended, but if you must, you should be careful to insert all nine bytes with interrupts disabled, to prevent an interrupt routine from calling OS_Mouse and possibly reading the wrong information.

OS_Byte &91 (145) – Get character from buffer

On entry : R1 = buffer number

On exit : R2 = character extracted
C = 1 if buffer empty

This call extracts the next character from a specified buffer. If the buffer was empty then the carry flag is set, and R2 will be invalid.

OS_Byte &98 (152) – Examine buffer status

On entry: R1 = buffer number

On exit: R2 = next character or preserved if empty
Carry reflect buffer empty status

This call returns the status of a specified buffer; the carry flag is set if the buffer is empty. If a character is available, it is returned in R2 but is not removed from the buffer.

OS_Byte &99 (153) – Insert character into buffer

On entry: R1 = buffer number (0 or 1)
R2 = character

On exit: R1 is undefined
R2 is undefined
C = 1 if buffer is full

This call enables characters to be inserted into one of the two input buffers as follows:

R1 = 0 insert character into the keyboard buffer
R1 = 1 insert character into the RS423 input buffer

If the current escape character (usually ASCII 27) is inserted, then appropriate action is taken, ie the escape condition is set, and an escape event is generated, if enabled. If the buffer was full and a character could not be inserted, then the carry flag is set on return.

These OS_Bytes are, in fact, just an interface to the vectored buffer routines described in the section **Software vectors**. Usually, the OS_Bytes are easier to use. However, there are times when it is preferable, or necessary (for example to read the bytes free in an input buffer) to use the vectors. They can be called directly using OS_CallAVector.

It is possible to change the operation of the machine by replacing these calls. In particular, you could write a module which OS_Claims all three buffer vectors, then replaces, say, the printer buffer with a much larger one, using memory claimed from the relocatable module area. Such a module could have its own configuration byte for the number of pages to use for the buffer, which it would claim on initialisation.

MISCELLANEOUS OS_BYTES

This section describes four OS_Bytes which don't really belong in any particular section of this guide.

OS_Byte &00 (0) – Display OS version information

On entry: R1 = 0 to display message
R1 <> 0 to return result

On exit: R1 = OS version number if R1 <> 0 on entry

If this is called with R1=0, an error is produced, and the text of the error shows the version number and creation date of the operating system. If it is called with R1<>0, then a version-dependent result is returned in R1.

OS_Byte &01 (1) – Write user flag

On entry: R1 = new value

On exit: R1 = old value

This OS_Byte accesses a location which is guaranteed to be unused by the OS. You can use this to pass results between programs. However, system variables provide much more versatile means of doing this. The byte may also be read and written using OS_Byte &F1 (241) below.

OS_Byte &F1 (241) – Read/write user flag

On entry: R1 = 0 or new value
R2 = &FF or 0

On exit: R1 = old value

OS_Byte &FO (240) – Read country flag

On entry: R1 = 0
R2 = &FF

On exit: R1 = country flag

This call returns the country value used by the international module.

CHARACTER OUTPUT

This chapter describes the ways in which characters can be sent to one or more of the Archimedes' output devices. There is one SWI which is central to all character output – OS_WriteC. When this routine is called, a character is sent to the current output streams. You can view an output stream as any device which expects characters; the screen is an obvious example. The properties of each output stream are described separately below.

CHARACTER OUTPUT ROUTINES

OS_WriteC (&00) – Write character

On entry: R0 = character to write

On exit: –

Vectored through WriteCV. This call sends the byte in R0 to all of the active output streams.

In addition to OS_WriteC, there are three related calls which send more than one character to the output streams (they use OS_WriteC to output each character). These calls are:

OS_WriteS (&01) – Write an in-line string

This routine writes a string which immediately follows the SWI instruction to the current output stream(s). The string is terminated by a zero-byte. Execution continues at the word after the end of the string.

On entry: –

On exit: –

OS_Write0 (&02) – Write an indirect string

This routine writes a zero-terminated string to the current output stream(s).

On entry: R0 points to the string to print

On exit: R0 points to the byte after the zero-byte

OS_NewLine(&03) – Write NewLine

This routine sends a line feed followed by a carriage return to the output stream(s) currently selected.

On entry: –

On exit: –

OS_NewLine is equivalent to two calls to OS_WriteI. For example:

```
SWI OS_WriteI + 10 ; VDU 10   ie linefeed
SWI OS_WriteC + 13 ; VDU 13   ie carriage return
```

may be replaced by:

```
SWI OS_NewLine ; VDU 10,13
```

The next two calls are available only on OS versions greater than 0.40.

OS_PrettyPrint (&44) – Print a formatted string

On entry: R0 = pointer to string to print

On exit: –

This call is only available under OS 0.40 and above. On entry, R0 points to a zero-terminated string. The string may contain normal printable characters, which are sent straight to OS_WriteC. Certain special characters may also be present:

- CR (ASCII 13) causes a newline to be generated
- TAB (ASCII 9) causes a tabulation to the next multiple of eight columns
- ASCII 31 is a 'hard space'.

Usually, `OS_PrettyPrint` will break a line at a space if the next word will not fit on the line; it will not do this at hard spaces.

`OS_Plot (&45)` – Perform a plot command

On entry: R0 = plot command code
 R1 = x co-ordinate
 R2 = y co-ordinate

On exit: –

This call is equivalent to a VDU PLOT command (see the chapter **THE VDU DRIVERS**). However, it is much more efficient as only one call is required (instead of six), and the VDU drivers are called directly instead of via the usual stream handling routines.

`OS_WriteN (&46)` – Write a counted string

This routine writes a string which is pointed to by R0. R1 contains a count of the number of bytes to write.

On entry: R0 = pointer to string to write
 R1 = number of bytes to write

On exit: –

If the VDU is the only active stream, this call uses the low-level VDU drivers directly, and is therefore much more efficient than using multiple calls to `OS_WriteC`. Also, because no special character is used to mark the end of the string, any VDU sequence may be sent.

`OS_WriteI (&100 – &1FF)` – Write an immediate byte

On entry: –

On exit: –

OS_WriteI writes the character contained in the bottom byte of the SWI number, using OS_WriteC. For example, to write a space character, you would use:

```
SWI OS_WriteI+ASC" " ; write a space
```

THE OUTPUT STREAMS

There are four principal output streams:

- The VDU stream; this means the Archimedes display
- The RS423 output stream; buffered
- The printer stream; buffered
- The *SPOOL file.

The RS423 port is a bi-directional port. Its input and output sides may be controlled independently. For example, you can transmit at a different baud rate from the one which you are using to receive. In this chapter, the output-specific calls to the RS423 port are covered. In the chapter **CHARACTER INPUT** both the input-specific and general RS423 calls are described.

There are several printer ports available. Unlike the output streams, where several may be used at once, only one printer may be active at any time. The comment 'buffered' next to the entries for the RS423 and printer means that the OS provides some buffering for these streams, which are emptied under interrupts. The *SPOOL file is buffered, but by the filing system.

An OS_Byte controls which of the output streams are enabled at any time.

OS_Byte &03 (3) – Specify output streams

On entry: R1 determines the output stream(s)

On exit: R1 contains the old stream specification
R2 is undefined

This call selects the device(s) to which all subsequent output will be sent. The output stream(s) are determined by which bits are set in R1 as follows:

Bit	Effect if set
0	Enables RS423 driver
1	Disables VDU driver
2	Disables VDU printer driver
3	Enables printer (independently of the VDU)
4	Disables spooled output
5	Calls VDUXV instead of VDU driver (see the chapter VDU DRIVERS)
6	Disables printer apart from VDU 1,n
7	Not used

The interpretations of all of these bits are described in subsequent sections. All bits are zero by default. This means that the VDU is enabled, the VDU printer driver is enabled, and the *SPOOL stream is enabled.

OS_Byte &EC (236) may also be used to read or set this variable.

OS_Byte &EC (236) – Read/write character destination status

On entry: R1 = 0 or new status
R2 = 255 or 0

On exit: R1 = previous status
R2 = value of next location (cursor key status)

THE RS423 OUTPUT STREAM

When bit 1 of the output stream's byte is set, characters sent to OS_WriteC are routed to the RS423 output port. In particular, they are inserted into the RS423 output buffer (buffer number 2), where they remain until removed by the interrupt routine dealing with RS423 transmission.

The RS423 buffer is 255 characters long. If there is nothing connected to the RS423 port, inserting more than 255 characters will result in the machine 'hanging' while it waits for a character to be removed to make space for the new character. An escape condition abandons this wait.

To set the speed at which RS423 characters are transmitted, OS_Byte &08 is used (see below).

OS_Byte &08 (8) – Write RS423 transmit rate

On entry: R1 = baud rate code

On exit: R1 is undefined
R2 is undefined

This call sets the RS423 baud rate for transmitting data as follows:

Value	Baud rate
0	9600
1	75
2	150
3	300
4	1200
5	2400
6	4800
7	9600
8	19200

The default rate is that set by *CONFIGURE Baud.

An event can be caused by the RS423 output stream: an output buffer empty event, when the last character is removed. See the section **Events** in the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS** for details.

THE VDU STREAM

Bit 1 of the streams byte disables the VDU stream if it is set. This prevents characters from appearing on the screen. It also disables all screen graphics etc. Finally, as control codes will not be acted on, it disables the VDU printer driver, (described in the next section).

Disabling the VDU, by setting this bit, is independent of the 'disable VDU drivers' control code (ASCII 21) described in the chapter **VDU DRIVERS**. The main difference is that the VDU printer driver will still work (if already enabled by ASCII 2) after an ASCII 21.

(The Archimedes VDU drivers are very sophisticated and provide a large number of facilities for drawing on the screen. They are described in detail in the next chapter.)

Bit 5 of the streams byte controls the VDUXV stream. If this bit is set, then characters that would usually be sent to the VDU drivers are sent instead to the routine on the VDU extension vector. This allows you to replace the VDU drivers, usually temporarily. The font manager uses this facility. See the section **Software vectors** for more details.

From the diagram below, you can see that bit 5 is only checked if bit 1 is clear. That is, the VDU extension vector is only used if the VDU is enabled, and if the VDUXV bit is set. Notice also that you can cause the character sent to VDUXV to be printed by setting the carry flag on return from the vector.

THE PRINTER STREAM

Three bits in the output stream's byte control whether a character is sent to the printer. In addition, a character may also be sent to the printer under the control of the VDU stream.

Bit 2 provides 'global' control over the printer. If this bit is set, then it is not possible for `OS_WriteC` to cause a character to be inserted into the printer buffer. If it is clear, then the character may or may not be sent to the printer, depending on the state of the other bits.

Bit 6 acts in a similar way: if it is clear, characters may be sent to the printer, but if it is set, they are stopped. There is one way of still getting characters to the printer if bit 6 is set; this is described below.

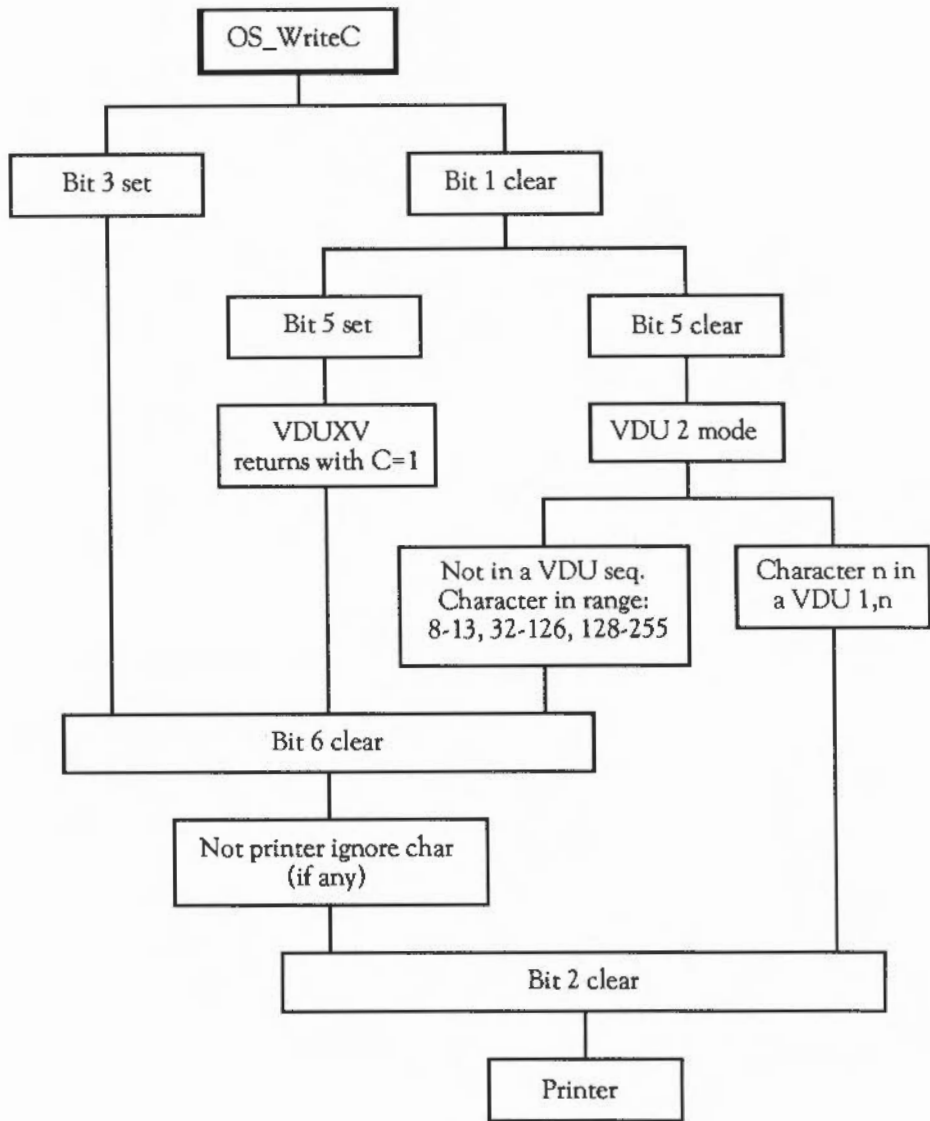
Assuming bits 2 and 6 are clear, then the simplest way of enabling the printer is by setting bit 3. When this is done, all characters sent to `OS_WriteC` (except the printer ignore character) will be inserted into the printer buffer too.

The most common way of controlling the printer is through the VDU driver. If the VDU stream is enabled (bit 1 of the stream's byte is clear), then sending the code ASCII 2 (**Ctrl**B) to OS_WriteC enables the 'VDU printer driver'. Once this is done, all printable characters, and some control characters, sent to the VDU stream will also go to the printer. Sending ASCII 3 (**Ctrl**C) to the VDU disables the copying of characters to the printer.

A further control code, ASCII 1 (**Ctrl**A), causes the next character to be sent to the printer (if enabled by **Ctrl**B), but not to the screen. All characters may be sent this way, including the control codes which are usually ignored by the VDU printer driver, and the printer ignore character.

If either bit 6 or bit 2 of the streams byte is set, then the VDU printer driver has no effect. The exception is when the character is preceded by a **Ctrl**A. In this case, bit 6 will not prevent the character from being sent, although bit 2 will.

The flow of control is summarised by the diagram below:



More details of the VDU printer driver control codes are given in the chapter **THE VDU DRIVER**.

Regardless of how a character gets to the printer, it is sent to the device controlled by the current printer type. This is set by another OS_Byte (see OS_Byte &05 (5)).

OS_Byte &05 (5) – Write printer driver type

On entry: R1 = driver type

On exit: R1 = previous driver type
R2 is undefined

The driver is determined as follows:

Value	Type
0	Printer sink
1	Parallel (Centronics) printer driver
2	RS423 output
3	User printer driver
4	Network printer driver
5 – 255	User printer driver

The old value is returned in R1. This call enables interrupts.

This call determines which printer driver type (and hence printer port) is selected for subsequent printer output. The default state is set by *CONFIGURE Print.

Note that if the RS423 port is selected as the printer, and the RS423 port is enabled by setting bit 0 of the stream's byte, then the character is inserted into both buffers. This means that eventually the character is printed twice (first from the RS423), so this practice is not recommended.

Instead of choosing an actual device type, for example a parallel printer driver, a 'printer sink' may be selected. This means that all characters sent to the printer are ignored.

The new destination type comes into effect only when all the current contents of the printer buffer have been sent to the previously-selected driver. This means that when you issue this OS_Byte, or the corresponding *FX command, the machine may appear to 'hang' until the current printer buffer's contents are cleared. (You may force this to happen by generating an escape condition.)

OS_Byte &F5 (245) may be used to read this variable, but not to set it (as it does not wait for the printer buffer to empty first). Because of this, it does not enable interrupts, and so may be used to read the printer type from within an interrupt routine.

OS_Byte &F5 (245) – Read printer driver type

On entry: R1 = 0
R2 = 255

On exit: R1 = previous value
R2 = value of next location (printer ignore character)

Like the RS423 stream, the printer stream may give rise to an output buffer empty event when the last character is removed.

The printer ignore character

The printer ignore character is discussed above. This is the character which is suppressed from the printer stream, unless the character got there via the VDU printer driver and was preceded by ASCII 1 (**Ctrl**A). There may be no printer ignore character set, in which case all characters are sent.

Below are descriptions of OS calls relating to the printer ignore character.

OS_Byte &06 (6) – Write printer ignore character

On entry: R1 = ASCII value of ignore character

On exit: R1 = old ignore character
R2 is undefined

The default value of the printer ignore character is set by *CONFIGURE Ignore. It may be changed temporarily using this OS_Byte, or by the associated command *IGNORE. The latter has the advantage that it also allows a 'no ignore' state to be set. See OS_Byte &B6 (182) below for details of this.

This variable may be read or set by OS_Byte &F6 (246) below.

OS_Byte &F6 (246) – Read/write printer ignore character

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 is undefined

The next OS_Byte relates to reading and setting the 'no ignore' state flag.

OS_Byte &B6 (182) – Read/write NOIGNORE state

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 determines the state
R2 is undefined

If the value read or written is ≥ 80 (ie has bit 7 set), then the printer ignore character is not used. If bit 7 is clear, then the current printer ignore character is filtered out.

The default setting of this flag is controlled by *CONFIGURE Ignore and may be changed temporarily using *IGNORE.

THE *SPOOL STREAM

The final stream belongs to the current *SPOOL file. When a *SPOOL file is opened, all characters subsequently displayed using OS_WriteC are also sent to that file, using the OS_BPut routine. This action continues until the file is closed, by another *SPOOL command. See the chapter **FILING SYSTEMS** for details of the *SPOOL command.

You can temporarily disable the *SPOOL file stream by setting bit 4 of the streams byte. This does not close the file, but simply prevents OS_WriteC from trying to send the character to file. To close the file, another *SPOOL command must be issued, or you can close the file directly by reading its handle using the OS_Byte documented below.

OS_Byte &C7 (199) provides direct control over the *SPOOL file, without the necessity of using the command line interpreter interface. It reads and writes the location which holds the handle of the current *SPOOL file. If this is zero, OS_WriteC makes no attempt to use the *SPOOL stream, as no file is open. So, to set up a *SPOOL file without using the command, you would perform the following steps:

- 1 Read the current handle
- 2 If it is non-zero, close the file (using OS_Find)
- 3 Open the new file for output (again using OS_Find)
- 4 Store the handle returned from the filing system in the *SPOOL handle location.

OS_Byte &C7 (199) – Read/write *SPOOL file handle

On entry: R1 = 0 or new handle
R2 = 255 or 0

On exit: R1 = previous handle
R2 = value of next location (Break or Escape status)

As noted in the previous chapter, the VDU drivers are quite complex, and deserve a chapter of their own. This chapter introduces the important concepts relating to the VDU, such as:

- screen modes
- window
- colour palette etc.

It then documents all the OS calls relating to the VDU. The most important one is `OS_WriteC`, as this is used in nearly all programs which have to make marks on the screen. Other calls, such as various `OS_Bytes` and `OS_Words`, are more concerned with returning information about the VDU.

There are three important aspects of VDU interaction which are not described in this chapter. These are the Font manager, the Window manager, and sprites. These are implemented as modules separate from the main OS, and are described in their own chapters.

VDU DRIVER CONCEPTS

Modes

The Archimedes has many different ways of displaying information on the screen, the exact number available depending on the type of monitor you have. They are all bit-mapped displays, in which one or more bits of screen memory control the colour of a dot, or pixel, on the screen. However, individual pixel control is not always available to the user: this is what differs between the text and graphic modes.

Two main characteristics distinguish the modes. The resolution of a mode relates to the number of pixels which can be displayed horizontally and vertically. Horizontal resolution can be 160, 320, 640, 1056, 1152 or 1280 pixels. Vertical resolution may be 256, 512, 864 or 976 pixels.

Secondly, the number of colours that can be displayed at once is determined by the number of bits used to store each pixel. This can be 1, 2, 4 or 8 bits, leading to 2, 4, 16 or 256 colours on the screen at once. Between them, the resolution and number

of colours determine the amount of screen memory used by a mode. This varies between 20K and 160K bytes.

A complete list of the available modes is given in the description of VDU 22 below.

Text and graphics

There are two distinct types of object that the VDU drivers can draw onto the screen ie text and graphics. It is as if the VDU drivers were split into two separate entities ie the text VDU and the graphics VDU. The text VDU deals with drawing text characters. Text characters are 8 by 8 patterns of pixels which are positioned on the screen at character-aligned positions. There is a text cursor which controls the position on the screen of the next character to be displayed. This is usually displayed as a flashing underline.

All text drawing is confined to an area know as the text window. This starts off as the whole screen (when a new mode is selected), but may be changed to any part of the screen, down to a single character cell. All scrolling is confined to this region, so it is sometimes called the scrolling window.

Various control codes are provided to affect the text VDU. Examples of such actions are:

- changing the colours in which text is drawn
- positioning and moving the text cursor
- clearing the window
- redefining the patterns which make up the displayed characters.

The graphics VDU controls the drawing of objects such as points, lines, circles, ellipses etc. Its facilities are not available in text-only modes. There is a separate window, the graphics or clipping window, which limits the area in which graphics may be drawn. There is also a graphics cursor, which is invisible and denotes the last point at which a graphics operation took place. Like the text VDU, the graphics VDU has its own colours etc.

There is a crossover point between the text and graphics VDUs. This is when the VDU is in 'VDU 5 mode'. In this mode, text characters are drawn as graphics

objects, using the current graphics cursor for positioning, and using the graphics colour. The advantage of this mode is that it enables characters to be drawn at any pixel alignment, and to be clipped to the graphics window (important when you use the Wimp environment). The disadvantage is that the characters take longer to draw and scrolling is not available.

The palette

Another important part of the VDU is the palette. As noted above, each mode may display 2, 4, 16 or 256 colours at once. These colours, however, are not fixed; they may be selected from 4096 actual colours. The palette is a table built in to the VDU hardware which determines the relationship between the colour number stored in the screen memory (the logical colours), and the actual colour information sent to the monitor (the physical colour).

The palette is programmed in terms of the intensity of the signal on each of the red, green and blue guns in a colour monitor. These intensities have 4 bits each, which gives twelve bits altogether, hence the 4096 physical colours. In fact, each logical colour can have 2 physical colours associated with it. These may be swapped at programmable rates, allowing flashing colours.

The palette also controls the colour of the border around the screen and the colour of the mouse pointer. These can be set independently of any other colour on the screen.

Screen RAM and banks

A certain amount of the machine's RAM is allocated for use by the screen. You can control how much is reserved using the *CONFIGURE ScreenSize command. Typically, 80K is reserved on a 0.5M byte machine and 160K on larger machines.

Because screen modes often require less memory than is actually reserved, there is room to have more than one 'bank' of screen RAM. For example, if 80K has been configured for the screen, there is enough room for four 20K screens. Only one can be displayed at once, of course. However, it is possible to tell the VDU drivers to write into one bank, while the hardware is displaying another. This allows updating

of the screen to occur 'behind the scenes', so that a completed screen can be instantaneously displayed.

The simplest form of banked screen RAM is the shadow mode concept. This involves two banks – the normal bank, and the so-called shadow bank. When you change mode, the *SHADOW command determines whether the normal bank or the shadow bank will be used. In addition, you can force the use of the shadow bank by adding 128 to the mode number, whether automatic shadow mode is in force or not.


Finally, a pair of OS_Bytes enables you to switch instantly amongst the current screen banks (however many there are), with independent control over what the hardware displays and what the VDU drivers use. An OS_Word gives finer control, enabling the start of the screen to be set to any address within the allocated screen area.

Cursor editing

Although the cursor editing facility isn't strictly part of the VDU drivers, its presence does have some interaction with the VDU. Usually there is only one text cursor, shown as a flashing underline. This is known as the output cursor, as it denotes where the next character will be output on the screen.

When you press one of the four cursor direction keys, cursor editing mode starts. There are now two cursors: the output cursor, which is now shown as a steady 'blob', and the input cursor, which is an underline flashing at twice the previous rate.

If you use OS_Byte &86 to read the text cursor position, or OS_Byte &87 to read the character underneath the cursor, the results always relate the flashing cursor, i.e. the output cursor normally, and the input cursor in cursor edit mode.

The final two effects to note about cursor editing mode are that it is not available in VDU 5 mode, and it is cancelled when you send an ASCII 13 (carriage return) to the VDU stream. This is usually done when you press  at the end of an input line.

Using OS_WriteC

This section describes all of the facilities that the OS_WriteC routine provides when the VDU stream is enabled. As shown in the last chapter, OS_WriteC is called with the character to display in R0. There are two types of character: printable and control. Printable characters are those with ASCII codes in the range 32 – 126 and 128 – 255. When these are sent to the VDU drivers, the pixel pattern corresponding to the character code is drawn onto the screen.

The way in which a printable character is drawn depends on whether VDU 5 is active or not. If it is not (the default state), the character is printed at the position denoted by the text cursor. Then the text cursor is moved on. At the end of the line, the cursor moves to the start of the next line, and if this is the bottom of the text window, the window is scrolled up. (In fact, this is the default action; you can control exactly what happens after a character is printed using one of the control sequences described below.)

In VDU 5 mode, printable characters are displayed at the graphics cursor, and the cursor is moved to the right by one character width. At the end of the line (which means the right edge of the graphics window in this case), the cursor is moved to the start of the next line. At the bottom of the screen, the cursor wraps round to the top. Again, this is the default behaviour. The direction of cursor movement, and indeed whether it moves at all, is under your control.

The codes which do not belong to printable characters, ie 0 – 31 and 127, are called control codes. These perform some action on the screen. Simple control codes are complete in themselves. For example ASCII 13 (carriage return) moves the cursor to the start of the current line. Others are more complex and have to be followed by one or more (up to nine) 'parameter' bytes. An example is ASCII 1, which is followed by a character to send to the VDU printer driver.

These multi-byte sequences aren't executed until all the required parameter bytes have been sent to the VDU driver. When the initial control code has been sent, subsequent bytes are 'queued'. That is, they are stored in memory until the last byte has been sent. When this has been done, the VDU drivers can access the queue and take the appropriate action.

Below are the descriptions of the VDU control sequences. They are given in terms of the BASIC VDU statement which you might use to execute them. Here is a brief reminder of the syntax of that statement:

VDU n sends ASCII code n to OS_WriteC. VDU m,n sends ASCII m followed by ASCII n.

VDU n; sends the number n as two bytes, first n MOD &100, then n DIV &100. This sends 16-bit numbers to the VDU drivers, eg co-ordinates in graphics commands.

VDU n| sends n as a single byte, followed by nine 0 bytes. This is used as shorthand in calls in which not all of the parameter bytes are needed. As nine is the largest number of bytes required by any VDU sequence, ending the command with '|' guarantees enough bytes to complete it. Any extra zeros are ignored by the VDU drivers.

Of course, as long as the correct characters are sent to the VDU, it doesn't matter how they get there. For example, the assembly language equivalent to VDU 12 (clear screen) is:

```
SWI "OS_WriteI"+12
```

The effect is the same in both cases.

VDU CONTROL SEQUENCES

VDU 0 – Null operation

VDU 0 does nothing. It is this that enables the '|' character in the VDU statement to work. Any of the nine zeros that are sent which aren't required by the current VDU command are 'swallowed up'.

VDU 1 – Next character to printer only

VDU 1 sends the next character to the printer only, provided that the printer has been enabled by VDU 2. Otherwise, the next character is ignored. This enables the

printer ignore character, and any other character which is not usually passed on by the VDU printer driver, to be sent to the printer through the VDU.

VDU 2 – Enable printer

VDU 2 enables the printer. After this call, most characters sent to the screen will also be sent to the printer port currently selected (see OS_Byte &05). Only characters in the following ranges are sent to the printer: 32 – 126, 128 – 255 (ie the printable characters), 8 – 13 (backspace, horizontal tab, line feed, vertical tab, form feed and carriage return, respectively). No multi-byte control sequences, except VDU 1, are sent to the printer.

Even if the VDU drivers are disabled (using VDU 21) the characters sent to the VDU drivers will still be sent to the printer although they will no longer affect the screen. However, if the VDU is disabled using OS_Byte &03, then VDU 2 printing will not take place.

The effect of VDU 2 can be cancelled using VDU 3.

You can determine whether VDU printing is enabled using OS_Byte &75 (117).

VDU 3 – Disable printer

VDU 3 cancels the effects of VDU 2 so that all subsequent printable characters are sent to the screen only.

VDU 4 – Split cursors

VDU 4 cancels VDU 5 mode. It causes all subsequent printable characters to be printed at the current text cursor position using the current text foreground and background colours. The text cursor is normally displayed (unless it has been disabled using VDU 23) and after each character has been printed the cursor moves on by one character. The direction of cursor movement is normally to the right but may be altered (using VDU 23).

After a character has been printed at the end of a row (or column if vertical printing is used) the cursor moves on to the start of the next screen line (or column),

scrolling the screen when there are no more rows (or columns). Cursor editing is allowed in this mode.

You can determine whether the cursors are split or joined using OS_Byte &75 (117).

VDU 5 – Join cursors

This enters VDU 5 mode. It links the text and graphics cursors and causes all subsequent printable characters to be printed at the current graphics cursor position, the top left of the character being placed there. Characters are displayed in the current graphics foreground colour using the current graphics action. The background colour is not altered.

After the character has been printed, the graphics cursor is moved by one character position. The direction of cursor movement is normally to the right but may be altered (using VDU 23). It moves to a new row (or column if vertical printing is being used) when necessary, or to the opposite corner of the graphics window if there are no more rows (or columns). Scrolling does not occur.

This command allows characters to be placed at any position on the screen, but means that the text is printed somewhat slower than when the cursors are split. In addition, each character is superimposed onto the existing text or graphics. Hence, printing a backspace character followed by a space moves the graphics cursor back by one character and then superimposes a space onto the character already there, thereby leaving it unaltered.

Cursor editing is not possible in this mode.

VDU 5 has no effect in text-only or teletext modes. In other modes it may be cancelled using VDU 4.

VDU 6 – Enable screen output

VDU 6 restores the functions of the VDU driver after it has been disabled by VDU 21. It causes all subsequent printable characters to be sent to the screen and control sequences to be obeyed.

You can determine whether the VDU is enabled or disabled using OS_Byte &75 (117).

VDU 7 – Bell

VDU 7 generates either the default bell sound (as specified by *CONFIGURE Loud/Quiet and *CONFIGURE SoundDefault) or the bell sound defined using OS_Bytes &D3 – &D6 (211 – 214).

VDU 8 – Back space

VDU 8 causes either the text cursor (by default command) or the graphics cursor (in VDU 5 mode) to be moved back one character position (ie in the negative X direction). This normally means moving it to the left but will be different if the direction of cursor movement is altered (using VDU 23,16).

If the cursor was at the start of a row (or column if vertical printing is used) then it is moved back to the end of the previous row (or column), scrolling the screen if necessary. It does not cause the last character to be deleted.

VDU 9 – Horizontal tab

VDU 9 causes either the text cursor (by default) or the graphics cursor (in VDU 5 mode) to be moved on one character position (ie in the positive X direction). This normally means moving it to the right but is different if the direction of cursor movement is altered (using VDU 23,16).

If the cursor was at the end of a row (or column if vertical printing is used) then it is moved on to the start of the next row (or column), scrolling the screen if necessary.

VDU 10 – Line feed

VDU 10 causes either the text cursor (by default) or the graphics cursor (in VDU 5 mode) to be moved on one line (ie in the positive Y direction). This normally means moving it down but is different if the direction of cursor movement has been altered (using VDU 23,16).

If the cursor was on the last line then the screen will be scrolled provided that scrolling is enabled.

VDU 11 – Vertical tab

VDU 11 causes either the text cursor (by default) or the graphics cursor (in VDU 5 mode) to be moved back one line (ie in the negative Y direction). This normally means moving it up but will be different if the direction of cursor movement has been altered (using VDU 23,16).

If the cursor was on the first line then the screen will be scrolled, if scrolling is enabled.

VDU 12 – Form feed/clear screen

VDU 12 clears either the current text window (by default) or the current graphics window (in VDU 5 mode) to the current text or graphics background colour respectively. In addition, the text or graphics cursor is moved to the text home position (see VDU 30).

When sent to a printer, this code generally causes a new page to be started.

VDU 13 – Carriage return

VDU 13 causes the text cursor (by default or the graphics cursor (in VDU 5 mode) to be moved to the negative X edge of the relevant window at the same Y value. The negative X edge is normally the left edge but it may be changed (using VDU 23,16).

When sent to a printer, this code generally causes the print head to move to the start of the current line. Additionally, some printers may also generate a line feed.

VDU 14 – Page mode on

VDU 14 causes the screen display to wait for **[Shift]** to be pressed before the next scroll and periodically thereafter. Normally, approximately 75% of the number of lines in the current window is scrolled before it waits again. The effects of the command may be cancelled using VDU 15.

OS_Byte &75 (117) may be used to determine whether paged mode is enabled. See also OS_Byte &D9 (217).

VDU 15 – Page mode off

VDU 15 cancels the effect of VDU 14 so that scrolling is unrestricted.

VDU 16 – Clear graphics window

VDU 16 clears the current graphics window to the current graphics background colour using the graphics background action. It does not affect the position of the graphics cursor.

VDU 17,c – Set text colour

VDU 17 is used to assign a logical colour to either the text foreground or text background colour according to the value of c, as follows:

Value	Colour
0 – 127	foreground
128 – 255	background (colour in range 0 – 127)

If the absolute value of the parameter lies outside the allowed set for the current mode, it is treated MOD (the number of colours) so that it lies within that range. For example, in mode 1, which allows four colours, the commands VDU 17,9 and VDU 17,5 are equivalent to VDU 17,1.

The interpretation of the 'c' parameter depends on the type of mode:

Colours c parameter meaning

2,4,16 Logical colour for that pixel
256 Bottom 6 bits of c provide colour information:

Bit 5	Blue High component
Bit 4	Blue Low component
Bit 3	Green High component
Bit 2	Green Low component
Bit 1	Red High component
Bit 0	Red Low component

This allows 64 different colours to be obtained. Each of these can be used in one of four different tints, giving 256 available shades. See VDU 23,17 for more details.

The current text colours may be read using OS_ReadVduVariables.

VDU 18,k,c – Set graphics colour and action

VDU 18 is used to define either the graphics foreground colour or the graphics background colour, and the way in which it is to be plotted on the screen.

The graphics plotting action is determined by 'k' as follows:

Value	Action
0	Overwrite colour on screen with c
1	OR colour on screen with c
2	AND colour on screen with c
3	EOR colour on screen with c
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND colour on screen with (NOT c)
7	OR colour on screen with (NOT c)
8 – 15	As 0 to 7, but background colour is transparent
16 – 31	Colour pattern 1 using action 0 – 15
32 – 47	Colour pattern 2 using action 0 – 15

48 – 63	Colour pattern 3 using action 0 – 15
64 – 79	Colour pattern 4 using action 0 – 15
80 – 85	Giant colour pattern (patterns 1 – 4 placed side by side)

The range 8 – 15 is used in the following circumstances:

- If a sprite has a 'mask', then plotting it using one of these actions causes the mask to be used.
- Where the mask has a 0 bit, nothing is plotted; where it has a 1 bit, the appropriate sprite colour is plotted. If an action in the range 0 – 7 is used, the sprite mask is ignored. See the chapter `SPRITES` for more details.

These actions are also used in colour pattern plotting. If a pixel in the pattern has the same colour as the current graphics background colour, it is not plotted but left transparent instead. (If the action is used when setting a background colour pattern, then the pixel is left unplotted if it has the same colour as the current graphics foreground colour.)

The graphics colour is determined by 'c' as follows:

Value	Meaning
0 – 127	Foreground colour specified
128 – 255	Background colour specified (colour in range 0 – 127)

If the absolute value of the parameter lies outside the allowed set for the current mode, it is altered so that it lies within the range.

Where 'k' has specified a colour pattern, then 'c' is used only to determine whether the pattern is used for the graphics foreground or background colour (depending on whether it is less than 128 or not).

The interpretation of the 'c' parameter depends on the type of screen mode. See the table for VDU 17 above for details.

The current graphics colours and actions may be read using `OS_ReadVduVariables`.

VDU 19,l,p,r,g,b – Set palette

VDU 19 defines the colour palette relationship. It causes a specified logical colour for either the screen, border or cursor to be represented by a given physical colour.

The action depends on the value of 'p' as follows:

p = 0 – 15	Logical colour l = actual colour p r, g and b are ignored
p = 16	Logical colour l = r units red g units green b units blue This sets both flash palettes for logical colour l
p = 17	Defines first flash palette for logical colour l
p = 18	Defines second flash palette for logical colour l
p = 24	Defines border colour = r units red g units green b units blue l is not used
p = 25	Define logical colour l (1 – 3) of cursor = r units red g units green b units blue

In the cases where 'p' is greater than 15, on adding 128 to it, you also set the 'supremacy' bit of the appropriate palette entry. This is used when the Archimedes' video is mixed with an external video source, to provide a superimposed image.

In all cases, the red, green and blue parameters have a range 0 – 255. However, as only the top four bits are significant, the 16 possible values are &0X, &1X,

&2X,... &FX, where X means 'don't care'. The lower nibble may be significant in future versions of the hardware – you should use 0 for now.

There are 16 palette registers, which means that in modes with one, two and four bits per pixel, there is a register available for each of the logical colours. Therefore, each can be assigned a physical colour by a simple one-to-one relationship.

By default (after a mode change or VDU 20), the palette is set up using a setting where 'p' is in the range 0 – 15. The settings for each mode are:

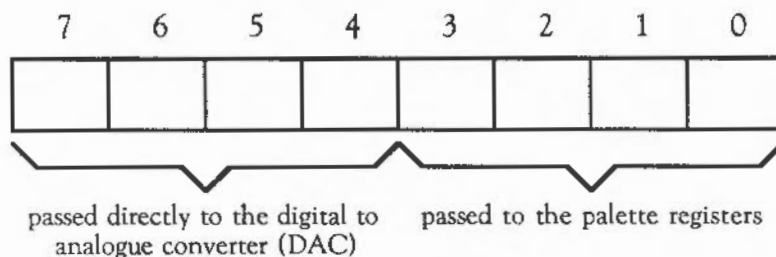
Logical colour Actual colour numbers for modes with:

	1 bit per pixel	2 bits per pixel	4 bits per pixel
0	0	0	0
1	7	1	1
2		3	2
3		7	3
4			4
5			5
6			6
7			7
8			8
9			9
10			10
11			11
12			12
13			13
14			14
15			15

The meanings of the 'p' type colours are:

Physical colour	Colour
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White
8	Black-white flashing
9	Red-cyan flashing
10	Green-magenta flashing
11	Yellow-blue flashing
12	Blue-yellow flashing
13	Magenta-green flashing
14	Cyan-red flashing
15	White-black flashing

In modes with eight bits per pixel the situation is more complex. A simple mapping of the logical colour to the physical colour via the palette is not possible. Instead, the eight bits of the logical colour are treated as two nibbles as follows:



- Bit 7 goes directly to the top bit of blue
- Bit 6 goes directly to the top bit of green
- Bit 5 goes directly to the second bit of green
- Bit 4 goes directly to the top bit of red

The default palettes are set to have the following effect:

- Bit 3 is sent to the second bit of blue
- Bit 2 is sent to the second bit of red
- Bit 1 is sent to the third bits of blue, green and red
- Bit 0 is sent to the fourth bits of blue, green and red

Hence the palette can only be used to produce subtle effects upon the colour; it does not have any effect upon the top (most significant) bits of any colour or the second bit of green. It can only control the second bits of blue and red and the white tint which is obtained by the settings of all three of the third and fourth (least significant) bits.

You can also set the palette using OS_Word &0C (12), and read the current palette using OS_Word &0B (11) and OS_ReadPalette.

VDU 20 – Restore default colours

VDU 20 restores the default palette for the current mode. It also resets the default text and graphics background colour to black, and the text and graphics foreground colour to white. The graphics foreground and background actions are set to 0 (overwrite). In 256-colour modes the tints are set to their default values (0 for background tints and &C0 for foreground ones).

VDU 21 – Disable screen display

VDU 21 prevents the VDU screen drivers performing any of their normal functions until a VDU 6 is issued. Any control sequences sent to the VDU drivers are queued in the usual way. Therefore, sending the code VDU 19 causes the next 5 characters to be treated as parameters for this (ignored) command.

For example, the sequence VDU 22,6 is treated as one whole command in the usual way and not as VDU 22 followed by VDU 6 which would re-enable the VDU drivers.

This command does not prevent characters from being sent to the VDU printer driver (if already enabled by a VDU 2), or any of the other output streams.

You can use OS_Byte &75 (117) to determine whether the VDU driver is currently enabled or disabled.

VDU 22,n – Change display mode

VDU 22 is used to select a screen mode. The modes available depend on the configured monitor type (see *CONFIGURE MonitorType in the section Operating system commands). The three types are:

- 0 'TV' type monitor
- 1 multi-sync monitor
- 2 64 KHz high-resolution monochrome monitor

Type 2 is only available on 400-series machines.

The bottom seven bits of 'n' are used to select the mode as follows:

MODE	Text col x row	Resolution hor x ver	Log.Cols	bits/pixel	Memory used
0	80 x 32	640 x 256	2	1	20K
1	40 x 32	320 x 256	4	2	20K
2	20 x 32	160 x 256	16	4	40K
3	80 x 25	Text only	2	2	40K
4	40 x 32	320 x 256	2	1	20K
5	20 x 32	160 x 256	4	2	20K
6	40 x 25	Text only	2	2	20K
7	40 x 25	TELETEXT	16		80K
8	80 x 32	640 x 256	4	2	40K
9	40 x 32	320 x 256	16	4	40K
10	20 x 32	160 x 256	256	8	80K
11	80 x 25	Text only	4	2	40K
12	80 x 32	640 x 256	16	4	80K
13	40 x 32	320 x 256	256	8	80K
14	80 x 25	Text only	16	4	80K
15	80 x 32	640 x 256	256	8	160K

16	132 x 32	Text only	16	4	132K
17	132 x 25	Text only	16	4	132K

There are four further modes which are only available for use on monitor types 0 and 1 (normal and multi-sync).

MODE	Text col x row	Resolution hor x ver	Log.Cols	bits/pixel	Memory used
18	80 x 64	640 x 512	2	1	40K
19	80 x 64	640 x 512	4	2	80K
20	80 x 64	640 x 512	16	4	160K
21	80 x 64	640 x 512	256	8	320K

The following two modes (and only these modes) are available if the configured monitor type is 2 (64 KHz line rate, 400-series machines only):

MODE	Text col x row	Resolution hor x ver	Log.Cols	bits/pixel	Memory used
22	160 x 122	1280 x 976	2	1	160K
23	144 x 54	Text only	2	1	160K

Notes on display modes

Modes 0 – 17 are available on type 0 and 1 monitors only. The refresh rate is 50Hz.

Modes 18 – 21 are available on type 0 and 1 monitors only. They are displayed at a 50Hz interlaced (25Hz frame) refresh rate on type 0 monitors. On type 1 monitors, modes 18 – 20 are displayed at a 50Hz refresh rate, and mode 21 has a 50Hz interlaced (25Hz frame) rate.

Type 2 monitors only may display modes 22 and 23, and these are the only modes available on type 2 monitors.

Versions of the OS up to and including 0.4 do not support modes 21 – 23, and only support modes 18 – 20 on type 0 and 1 monitors.

If an attempt is made to select a mode which is not appropriate to the current monitor type (or OS version), a suitable mode for that monitor is used. For example, an attempt to select mode 22 on a type 0 monitor will result in mode 0 being used.

If 128 is added to the mode number, the so-called shadow bank is used. Any display mode may have several banks of memory available. The number of banks depends on the size of the screen memory (as allocated by *CONFIGURE ScreenSize) and the size of the current mode. For example, if 160K is allocated, and 20K is used for the display, eight banks are available.

Usually, bank 1 is used. However, if 128 is added to the mode number, or a *SHADOW command has been issued, bank 2 is used after a mode change. Shadow memory can only be used if ScreenSize is at least twice the memory for the required mode.

The other banks may be accessed using OS_Byte &70 – &71 (112 – 113).

The mode command causes the following actions:

- Cursor editing is terminated if currently in use
- VDU 4 mode is entered
- The text and graphics windows are restored to their default values
- The text cursor is moved to its home position
- The graphics cursor is moved to (0,0)
- The graphics origin is moved to (0,0)
- Paged mode is terminated if currently in use
- The logical-physical colour map is set to the new mode's default
- The text and graphics foreground colours are set to white

- The text and graphics background colours are set to black (colour 0)
- The colour patterns are set to their defaults for the new mode
- The dot pattern for dotted lines is reset to &AAAAAAA
- The dot pattern repeat length is reset to 8
- The screen is cleared to the current text background colour.

The current screen mode may be read using OS_Byte &87 (135).

VDU 23 – Miscellaneous commands

VDU 23 is a multi-purpose command taking nine parameters, of which the first identifies a particular function. Each of the available functions is described below. Eight additional parameters are required in each case, though often most of these are ignored. This enables you to use 'l' as shorthand in VDU statements, eg:

```
VDU 23,0,10l
```

instead of

```
VDU 23,0,10,0,0,0,0,0,0,0
```

```
VDU 23,0,n,m,0,0,0,0,0
```

If n = 8, this sets the interlace as follows:

m = 0	sets the screen interlace state to the opposite of the current *TV setting
m = 1	sets the screen interlace state to the current *TV setting
m = &80	turns the screen interlace off
m = &81	turns the screen interlace on

If n = 10 or 11, this controls the height of the cursor on the screen and its appearance.

$n = 10$ m defines the start line for the cursor and its appearance:

Bits 0 – 4 define the start line (0 being the top)

Bits 5 – 6 define its appearance as follows:

Bit 6	Bit 5	Meaning
0	0	Steady
0	1	Off
1	0	Fast flash
1	1	Slow flash

$n = 11$ m defines the end line for the cursor.

The bottom line is 7 for 32-line modes, 9 for 25-line modes, and 19 for mode 7.

`VDU 23,1,n,0,0,0,0,0,0`

VDU 23,1 controls the appearance of the cursor on the screen depending on the value of n :

Value	Meaning
0	stops the cursor appearing
1	makes the cursor re-appear
2	makes the cursor steady
3	makes the cursor flash

The effect of this call is cancelled when cursor editing occurs. The effect of the previous call is not changed by cursor editing. See also SWI OS_RemoveCursors and SWI OS_RestoreCursors.

VDU 23,2 – 5,n1,...,n8

VDU 23,2 – VDU 23,5 are used to define the four colour patterns:

VDU 23,2	sets pattern 1
VDU 23,3	sets pattern 2
VDU 23,4	sets pattern 3
VDU 23,5	sets pattern 4

Each of the integers n1 to n8 defines one row of the pattern, n1 being the top row and n8 being the bottom. For a given parameter, the logical colours of the pixels in each row depend upon the number of colours available in the current screen mode and which pattern mode is used. There are two available pattern modes. The default is the Master 128-compatible mode in which a parameter is decoded as it would be on a BBC Master-series microcomputer. The other is the Archimedes mode which decodes the values in a simpler fashion. To change to this mode use VDU 23,17,4,11.

In the Master 128-compatible mode, if the bit settings in each of n1 to n8 are denoted by 76543210, then the logical colours of the pixels in each row (from left to right) are:

No. of colours	Bits per pixel	No. of pixels in pattern	Logical colours
2	1	8	7,6,5,4,3,2,1,0
4	2	4	73, 62, 51, 40
16	4	2	7531, 6420

For example in modes with four bits per pixel, bits 7, 5, 3 and 1 of the 'n' parameter control the logical colour of the left-hand pixel, and bits 6, 4, 2 and 0 control the right-hand pixel. To set the left pixel to colour 2 (green by default) and the right one to colour 7 (white), the colours are combined as follows:

pixel 1 colour

green (2)
0 0 1 0

Bit 7 6 5 4 3 2 1 0

Left pixel 0 0 1 0

Right pixel 0 1 1 1

Result 0 0 0 1 1 1 0 1 = &1D

Resulting value = &1D or 29

pixel 2 colour

white (7)
0 1 1 1

Whereas in modes with two bits per pixel the method is:

pixel 1 colour

yellow (2)
1 0

Bit

Pixel 1

Pixel 2

Pixel 3

Pixel 4

Result

Resulting value = &B6 or 182

pixel 2 colour pixel 3 colour

red (1) white (3)
0 1 1 1 1 0

7 6 5 4 3 2 1 0

1 0

0 1

1 1

1 0

1 0 1 1 0 1 1 0 = &B6

pixel 4 colour

yellow(2)

In Archimedes mode, the bits 76543210 of a parameter define the pattern as follows:

No. of colours	Bits per pixel	No. of pixels in pattern	Logical colours
2	1	8	0,1,2,3,4,5,6,7
4	2	4	10, 32, 54, 76
16	4	2	3210, 7654

For example, in modes with four bits per pixel, the colour of the left-hand pixel is formed from bits 3, 2, 1 and 0 of the 'n' parameter, and the colour of the right-hand pixel comes from bits 7, 6, 5 and 4 of the parameter. So, if the pixels are to be logical colours 2 and 7 again, the colours are combined as follows:

pixel 1 colour	pixel 2 colour
green (2)	white (7)
0 0 1 0	0 1 1 1
Bit	7 6 5 4 3 2 1 0
Left pixel	0 1 1 1
Right pixel	0 0 1 0
Result	0 1 1 1 0 0 1 0
Resulting value = &72 or 114	

Notice that the pixel colours on the left, as displayed, are derived from the bits on the right, as written down, and vice versa.

In modes with two bits per pixel the method is:

pixel 1 colour	pixel 2 colour	pixel 3 colour	pixel 4 colour
----------------	----------------	----------------	----------------

yellow (2)	red (1)	white (3)	yellow(2)
1 0	0 1	1 1	1 0

Bit	7	6	5	4	3	2	1	0
-----	---	---	---	---	---	---	---	---

Pixel 4	1	0						
Pixel 3			1	1				
Pixel 2				0	1			
Pixel 1						1	0	

Result	1	0	1	1	0	1	1	0
--------	---	---	---	---	---	---	---	---

Resulting value = &B6 or 182

VDU 23,6,n1,...,n8

VDU 23,6 sets the dot-dash line style used by dotted line PLOT commands (see also VDU 25 and OS_Byte &A3 (163)).

Each of the integers n1 to n8 defines eight elements of the line style, n1 being at the start and n8 at the end. The bits in each byte are read from most significant to least significant, each 1-bit indicating a dot and each 0-bit a space. The default is &AAAAAAAA (alternating dots and spaces) with a repeat length of eight (so only n1 is used).

VDU 23,7,m,d,z,0,0,0,0

VDU 23,7 allows the current text window or whole screen to be scrolled directly in any direction without moving the cursor. The values of m, d and z determine the area to be scrolled, the direction of scrolling and the amount of scrolling as follows:

m = 0	scroll the current text window
m = 1	scroll the entire screen

d = 0	scroll right
d = 1	scroll left
d = 2	scroll down
d = 3	scroll up
d = 4	scroll in positive X direction
d = 5	scroll in negative X direction
d = 6	scroll in positive Y direction
d = 7	scroll in negative Y direction
z = 0	scroll by one character cell
z = 1	scroll by one character cell vertically or one byte horizontally

If z=1, the horizontal movement depends on the number of colours in the current mode as follows:

Number of colours	Number of pixels moved
2	8
4	4
16	2
256	1

VDU 23,8,t1,t2,x1,y1,x2,y2,0,0

VDU 23,8 causes a block of the current text window to be cleared to the text background colour. The parameters t1 and t2 indicate base positions relating to the start and end of the block to be cleared respectively:

Value	Meaning
0	top left of window
1	top of cursor column
2	off top right of window
4	left end of cursor line
5	cursor position
6	off right of cursor line

8	bottom left of window
9	bottom of cursor column
10	off bottom right of window

References to 'left', 'up' and so on are dependent upon the cursor movement control set by VDU 23,16. 'Off' means 'one character beyond (in the positive x direction)'. The effects of other values, ie 3, 7 and any number over 10, are undefined.

The parameters x1,y1 and x2,y2 are displacements from the positions specified by t1 and t2 and determine the start and end of the block:

x1	Displacement from t1 in x direction
y1	Displacement from t1 in y direction
x2	Displacement from t2 in x direction
y2	Displacement from t2 in y direction

The result is undefined if the absolute values defining the start and end of the block produce values outside the range -128 to 127. If the end point of the block lies before the start point then no clearing takes place.

The action of this command can be viewed as equivalent to moving the text cursor to the start of the block, then printing spaces until the end of the block is reached (but without printing a space at the last position).

VDU 23,9,n l

VDU 23,9 sets the flash time for the first flashing colour. The length is determined by the value of n as follows:

n = 0	sets an infinite duration (steady colour)
n <> 0	sets the duration to n VSYNCs

A VSYNC is the time between refreshes of the screen display. It varies between display modes and countries. In the UK for modes 0 - 16 it is approximately 50Hz.

This command is equivalent to OS_Byte &09.

VDU 23,10,nI

VDU 23,10 sets the flash time for the second flashing colour. The length is determined by the value of n as follows:

n = 0 sets an infinite duration (steady colour)
 n <> 0 sets the duration to n VSYNCs

This command is equivalent to OS_Byte &0A.

VDU 23,11,0,0,0,0,0,0,0

VDU 23,11 selects the Master-128 compatible pattern mode and causes the four colour patterns to be reset to their defaults for the current screen mode. With the default logical-physical map, these defaults are:

Modes 0,4,18

1 – Dark grey	2 – Grey	3 – Light grey	4 – Hatching
10101010	10101010	11111111	00010001
00000000	01010101	01010101	00100010
10101010	10101010	11111111	01000100
00000000	01010101	01010101	10001000

Modes 1,5,8,19

1 – Red-orange	2 – Orange	3 – Yel-orange	4 – Cream
2121	2121	2222	2323
1111	1212	1212	3232
2121	2121	2222	2323
1111	1212	1212	3232

Modes 2,9,12,20

1 – Orange	2 – Pink	3 – Yel-green	4 – Cream
21	61	32	37
12	16	23	73
21	61	32	37
12	16	23	73

All the patterns repeat after four rows, so only the first four are shown.

VDU 23,12 – 15,n1,n2,n3,n4,n5,n6,n7,n8

VDU 23,12 – 15 are used to define the four colour patterns in a simpler way than that provided by VDU 23,2 – 5. The limitation is that you can only set a two-by-four pattern of pixels.

VDU 23,12 sets colour pattern 1
VDU 23,13 sets colour pattern 2
VDU 23,14 sets colour pattern 3
VDU 23,15 sets colour pattern 4

The pixels of the top row of the resulting pattern are assigned alternating logical colours n1 and n2, those of the next row have colours n3 and n4 etc. For example, to set up the following pattern in mode 1:

RedYel	12
WhtRed	31
BlkRed	01
WhtYel	32

the required sequence is VDU 23,12,1,2,3,1,0,1,3,2

VDU 23,16,x,y|

VDU 23,16 gives control of the movement of the cursor after a character has been printed. This movement is under the control of a byte of flags. VDU 23,16 replaces the byte by:

((current byte) AND y) EOR x

The interpretation of the flags is as follows:

7	bit 7 = 0	Normal.
	bit 7 = 1	Undefined.
6	bit 6 = 0	In VDU 5 mode, cursor movements beyond the current edge of the window cause special actions. For example, they generate newlines at the end of the line.
	bit 6 = 1	In VDU 5 mode, cursor movements beyond the edge of the window do not cause special actions.
5	bit 5 = 0	Cursor moves in the positive X direction after the character is printed. If this results in the cursor moving beyond the edge of the window, the settings of bits 6, 4 and 0 define the action which is taken.
	bit 5 = 1	Cursor does not move after the character is printed.
4	bit 4 = 0	When a cursor movement in the Y direction results in the cursor moving beyond the window edge, the window is scrolled if in VDU 4 mode. If in VDU 5 mode, the cursor moves to the opposite edge of the window.
	bit 4 = 1	When a cursor movement in the Y direction results in the cursor moving beyond the window edge, the cursor is always moved to the opposite edge of the window.
3	bit 3 = 0	X direction is horizontal, Y direction is vertical.

- bit 3 = 1 X direction is vertical, Y direction is horizontal.
- bit 2 = 0 Positive vertical direction is down.
- bit 2 = 1 Positive vertical direction is up.
- bit 1 = 0 Positive horizontal direction is right.
- bit 1 = 1 Positive horizontal direction is left.
- bit 0 = 0 Disables the scroll-protect option. When printing a character in VDU 4 mode results in the cursor moving beyond the edge of the window, the cursor is instead moved to the negative X edge of the window and one line in the positive Y direction.
- bit 0 = 1 Enables the scroll protect option. When printing a character in VDU 4 mode results in the cursor moving beyond the edge of the window, a 'pending newline' is generated. It is actually executed just before the next character is printed, provided that it has not been deleted or executed by another cursor control code. For example VDU 127 would cancel it; VDU 9 would execute it.

VDU 23,17,n,m l

VDU 23,17,0-3 is used to set the tint for a colour in the 256-colour modes. The 'n' parameter determines which colour is set, as follows:

Value	Colour
0	sets the tint for the text foreground colour
1	sets the tint for the text background colour
2	sets the tint for the graphics foreground colour
3	sets the tint for the graphics background colour

The VDU 17-18 commands control the top two bits of blue, green and red independently of each other. This command allows the bottom two bits to be

controlled. However, they cannot be set independently. The least significant bits must either all be set or all clear. Hence it determines the amount of white tint given to the colour.

The value of the tint is given by the top two bits of 'm':

Value	Tint
&00	Bit 0 and bit 1 clear (darkest)
&40	Bit 0 set and bit 1 clear
&80	Bit 1 set and bit 0 clear
&C0	Bit 0 and bit 1 set (lightest)

When a pixel is plotted the following occurs, in terms of the actual logical colour stored in the screen memory: the bottom six bits of the colour number (set by VDU 17 – 18) are shifted up by two bits, giving bits 2 – 7 of the colour byte; the appropriate tint value is shifted down by six bits, into bits 0 and 1, and the two parts are then combined.

VDU 23,17,4,m | chooses which set of default colour patterns are used, depending on the value of 'm':

Value	Mode
0	Use 6502 BBC Micro compatible colour patterns
1	Use native colour patterns

VDU 23,17,5 | exchanges the current text foreground and background colours. After the first VDU 23,17,5 | subsequent characters printed are in inverse video. After the second VDU 23,17,5 | subsequent characters printed are of normal appearance.

VDU 23,18..25 |

VDU 23,18 – VDU 23,24 are reserved for future Acorn extensions.

VDU 23,25 – 26,n1,n2,n3,n4,n5,n6,n7,n8

These calls are provided by the Font Manager. See the chapter **THE FONT MANAGER** for details.

VDU 23,27,m,n,0,0,0,0,0

This call is provided by the Sprite Manager. See the chapter **SPRITES** for details.

VDU 23,28..31,n1,n2,n3,n4,n5,n6,n7,n8

VDU 23,28 to VDU 23,31 are reserved for use by applications programs.

VDU 23,32 – 255,n1,n2,n3,n4,n5,n6,n7,n8

VDU 23,32 to VDU 23,255 redefine the printable ASCII characters. The redefined character depends on the value of the second parameter. For example, VDU 23,65 redefines the character whose ASCII code is 65, ie capital A. The parameters n1 to n8 are integers representing the eight rows of the character to be redefined, n1 being the top row and n8 the bottom row. Each bit of a value represents one pixel of the corresponding row, with a '1' indicating that the corresponding pixel is to be plotted in the foreground colour and a zero that it is to be plotted in the background colour (or not at all in the case of VDU 5 mode printing). The most significant bit of the byte corresponds to the left-hand pixel of its row, and the others follow linearly.

Although the delete character (ASCII 127) can be redefined, redefining has no effect as it cannot be displayed.

You can read the pattern for a given character using `OS_Word &0A`.

VDU 24,x1;y1;x2;y2; – Define graphics window

VDU 24 allows the user to define a graphics window. Any graphics objects which are drawn (including VDU 5 mode and fancy-font characters) and which lie outside this window are clipped to the edges of the window. The four parameters define the left, bottom, right and top boundaries of the window respectively, relative to the current graphics origin (the bottom left of the screen, by default). The window which you

are defining must lie within the screen boundaries, otherwise the command is ignored.

Use `OS_ReadVduVariables` to discover the size of the current graphics window.

VDU 25,k,x;y; – General PLOT command

VDU 25 is a multi-purpose graphics plotting command. The first parameter defines a particular function. The other parameters are the x co-ordinate and the y co-ordinate. They are relative either to the current graphics origin, or to the last point visited, depending on the value of 'k'.

The bottom three bits of 'k' determine the manner in which the plot is to be performed:

(k AND 7) =	0	move cursor relative (to last graphics point visited)
	1	plot relative using current foreground colour
	2	plot relative using logical inverse colour
	3	plot relative using current background colour
	4	move cursor absolute (ie move to actual co-ordinates given)
	5	plot absolute using current foreground colour
	6	plot absolute using logical inverse colour
	7	plot absolute using current background colour

The remaining bits of 'k' determine the action to be performed:

k =	0 – 7	Solid line including both end points
	8 – 15	Solid line excluding the final point
	16 – 23	Dotted line including both endpoints, pattern restarted
	24 – 31	Dotted line excluding the final point, pattern restarted
k =	32 – 39	Solid line excluding the initial point
	40 – 47	Solid line excluding both end points
	48 – 55	Dotted line excluding the initial point, pattern continued
	56 – 63	Dotted line excluding both end points, pattern continued

k =	64 – 71	Point Plot
	72 – 79	Horizontal line fill (left and right) to non-background
	80 – 87	Triangle fill
	88 – 95	Horizontal line fill (right only) to background
k =	96 – 103	Rectangle fill
	104 – 111	Horizontal line fill (left and right) to foreground
	112 – 119	Parallelogram fill
	120 – 127	Horizontal line fill (right only) to non-foreground
k =	128 – 135	Flood to non-background
	136 – 143	Flood to foreground
	144 – 151	Circle outline
	152 – 159	Circle fill
k =	160 – 167	Circular arc
	168 – 175	Segment
	176 – 183	Sector
	184 – 191	Block copy/move *
k =	192 – 199	Ellipse outline
	200 – 207	Ellipse fill
	208 – 215	Graphics Characters
	216 – 223	Reserved for Acorn Expansion
k =	224 – 231	Reserved for Acorn Expansion
	232 – 239	Sprite Plot – see the chapter SPRITES
	240 – 247	Reserved for User programs
	248 – 255	Reserved for User programs

* The eight codes in the range 184 – 191, which perform a block copy/move, have the following meanings:

184	Move relative
185	Relative rectangle move
186	Relative rectangle copy
187	Relative rectangle copy

188	Move absolute
189	Absolute rectangle move
190	Absolute rectangle copy
191	Absolute rectangle copy

Some of the objects require several points to be specified in order to define the shape completely. The last plot does the actual drawing. The sequences of moves and draws required for each type are:

Shape	Sequence of moves
Line	Move to one endpoint. Plot line to other endpoint.
Triangle	Move to first vertex. Move to second vertex. Plot triangle to last vertex.
Rectangle	Move to one corner. Plot rectangle to diagonally-opposite corner.
Parallelogram	Move to first corner. Move to second corner. Plot parallelogram to third corner. The fourth corner is derived from the other three.
Circle	Move to centre. Plot circle to point on the circumference.
Arc, segment, sector	Move to centre of circle. Move to start of arc. Plot to a point on the line from the centre to the end of the arc. Arcs, etc, are always drawn counter-clockwise.
Block copy/move	Move to one corner of source rectangle. Move to diagonally-opposite corner of source rectangle. Plot block copy/move to lower left of destination rectangle.
Ellipse	Move to centre. Move to intersection of ellipse circumference and centre's Y co-ordinate. Plot ellipse to highest or lowest point on the ellipse.

VDU 26 – Restore default windows

VDU 26 causes the text and graphics windows to be reset to their default states, ie both become the full screen. In addition, the command resets the graphics origin to (0,0), moves the graphics cursor to (0,0) and moves the text cursor to its home position. Hardware scrolling of the text window is initiated.

VDU 27 – No operation

VDU 27 has no effect.

VDU 28,lx,by,rx,ty – Define text window

VDU 28 defines (or redefines) a text window. The parameters are integers specifying the boundary of the window as follows:

lx =	left-most x column
by =	bottom-most y row
rx =	right-most x column
ty =	top-most y row

If the command attempts to define a window which extends outside the screen boundaries, has lx greater than rx, or has by less than ty, it will have no effect. The smallest possible window is one character.

You can read the size of the current text window using OS_ReadVduVariables.

VDU 29,x;y; – Set graphics origin

VDU 29 defines the point specified as the origin to be used for all subsequent graphics output using VDU 25 commands, and for the graphics window defined by VDU 24. The parameters are the two pairs of bytes specifying the absolute x and y co-ordinates of the new origin.

- *Note:* changing the graphics origin does not alter the position of the graphics window on the screen. The window's co-ordinates in terms of the origin therefore effectively change after a VDU 29.

You can read the position of the current origin using `OS_ReadVduVariables`.

VDU 30 – Home text cursor

VDU 30 moves the text cursor to its 'home' position. This is normally the top left of the window but may be changed (using VDU 23,16). In VDU 5 mode the graphics cursor is moved instead. It may have an offset of up to seven pixels out of the corner along one or both of the axes to allow for the height or width of the character depending on the direction of character printing.

VDU 31,x,y – Position text cursor

VDU 31 moves the text cursor to a specified x and y co-ordinate on the screen. The parameters x and y are the column and row numbers.

In VDU 4 mode, x and y are given relative to the text 'home' position which is at (0,0). If the position lies outside the text window, nothing happens, unless the scroll protect option is enabled and the x co-ordinate is just beyond the positive X edge of the window. In this case, the text cursor is moved to position (x-1,y) and a pending newline is generated.

In VDU 5 mode the graphics cursor is moved to its 'home' position plus 8*x pixels in the positive X direction, plus 8*y pixels in the positive Y direction. It is possible to move the cursor outside the graphics window in VDU 5 mode.

You can read the position of the text cursor using `OS_Byte &86 (134)`.

VDU 127 – Delete

Unless the previous use of VDU 23,16 indicates that no cursor movement is to take place after character printing, the cursor is moved backwards as if by VDU 8. Then the character under the cursor is deleted by overprinting it with a space (in VDU 4 mode) or a solid block of graphics background colour (in VDU 5 mode). These space and solid block characters are selected from the 'hard' (rather than the 'soft') font, so redefining these characters will not change the results.

THE VDU OS_BYTES

OS_Byte &09 (9) – Write duration of first colour

On entry: R1 = duration

On exit: R1 = old value of duration
R2 is undefined

This call sets the duration of the first flash colour.

Flashing colours are displayed as a sequence of two alternating colours. By default, each colour is displayed for 25 video frames at a time, which is approximately 0.5 seconds in the UK. This command allows you to alter the duration for which the first colour is displayed as follows:

Value	Meaning
0	Set an infinite duration (first colour constantly displayed)
n	Set the duration to n video frames (approximately n/50 seconds)

This variable may also be set using VDU 23,9. It may be read (but not set) by OS_Byte &C3 (195) (see below).

OS_Byte &C3 (195) – Read duration of first colour

On entry: R1 = 0
R2 = 255

On exit: R1 = previous value
R2 = value of next location (keyboard auto repeat delay)

OS_Byte &0A (10) – Write duration of second colour

On entry: R1 = duration

On exit: R1 = old value of duration
R2 is undefined

This call sets the duration for the second flash colour.

This variable may also be set using VDU 23,10. It may be read (but not set) by OS_Byte &C2 (194).

OS_Byte &C2 (194) – Read duration of second colour

On entry: R1 = 0
R2 = 255

On exit: R1 = previous value
R2 = value of next location (duration of first colour)

OS_Byte &13 (19) – Wait for vertical sync (vsync)

On entry: –

On exit: R1 is undefined
R2 is undefined

The video display frame is drawn approximately fifty times a second in the UK. This call synchronises a software routine with the signal produced when the electron beam reaches the bottom of the displayed area of the picture (ie the start of the border).

From the time that the beam reaches the bottom of the display until the next frame starts to be displayed, the electron beam is either 'blanked' or shows the border colour. This means that you have this time (3.4ms in modes 0 – 17, 0.7ms in modes 18 – 20) to redraw the screen. Because the beam is blanked while this redrawing is taking place, there is no flicker.

If 3ms is not enough time to produce a flicker-free update of the screen, you should consider using more than one bank of screen memory and switching between them (using OS_Byte &70 – &71 for example).

OS_Byte &14 (20) – Reset font definitions

On entry: –

On exit: R1 is undefined
R2 is undefined

The shape of the character displayed when printing ASCII codes 32 – 255 may be redefined using the VDU 23 command. Any such changes remain in force until the next hard reset. This command may be used to restore the default character definitions for ASCII codes in the range 32 – 127.

Note that you should really only redefine characters in the range 128 – 159. This is because all of the other printable characters have ‘standard’ meanings which should be preserved for use in applications such as word processors.

See OS_Byte &19 (25) for details on how to restore the other codes or how to restore a smaller selected group.

OS_Byte &19 (25) – Reset group of font definitions

On entry: R1 = group to restore

On exit: R1 is undefined
R2 is undefined

All ASCII characters between 32 and 255 may be redefined using the VDU 23 command. This call restores all or a particular group of characters to their default settings according to R1, as follows:

Value	Meaning
0	Restore characters 32 – 255
1	Restore characters 32 – 63
2	Restore characters 64 – 95
3	Restore characters 96 – 127
4	Restore characters 128 – 159

5	Restore characters 160 – 191
6	Restore characters 192 – 223
7	Restore characters 224 – 255

OS_Byte &70 (112) – Write VDU driver screen bank

On entry: R1 = bank number

On exit: R1 = old bank number
R2 is undefined

This call selects the bank of screen memory which is to be used by the VDU drivers according to R1, as follows:

Value	Bank
0	Default for the current screen mode (1 or 2)
n	Select bank 'n'

The maximum value for 'n' is $(\text{ScreenSize})/(\text{ModeSize})$, where ScreenSize is the *CONFIGURED screen size, and ModeSize is the size of the current mode. For example, in mode 0, a 20K mode with 160K set aside for the screen makes eight banks available, so 8 is the maximum value for 'n'.

The default bank for a non-shadow mode is bank 1; for a shadow mode it is bank 2. OS_Byte &FA may be used to read the bank number without writing it (see below).

OS_Byte &FA (250) – Read VDU driver screen bank number

On entry: R1 = 0
R2 = 255

On exit: R1 = screen bank used by VDU drivers
R2 = next location (display screen bank)

OS_Byte &71 (113) – Write display hardware screen bank

On entry: R1 = bank number

On exit: R1 = old bank number
R2 is undefined

This call selects the bank of screen memory which is to be used by the display hardware according to R1:

Value	Bank
0	Default for the current screen mode
n	Select bank n

The bank may be read (but not set) using OS_Byte &FB (see below).

OS_Byte &FB (251) – Read display screen bank number

On entry: R1 = 0
R2 = 255

On exit: R1 = screen bank used by the display
R2 is undefined

OS_Byte &72 (114) – Write shadow/non-shadow state

On entry: R1 = shadow state

On exit: R1 = old value
R2 is undefined

This call determines whether future MODE commands will be forced into the shadow state, depending on R1:

Value	Meaning
0	Modes will be shadow
1	Modes will be non-shadow

Shadow state requires twice the amount of RAM than the equivalent non-shadow mode since two copies of the screen are stored in memory. OS_ByteS &70 (112) and &71 (113) control the use of the banks.

To select a shadow state temporarily when in non-shadow mode, you can use the MODE 128+n convention. Future MODE commands will not be influenced by this.

OS_Byte &75 (117) – Read VDU status

On entry: –

On exit: R1 = status byte

This call returns the content of the VDU status byte. This byte gives information on the way in which characters are output according to their bit settings:

Bit	Status when set
0	Printer output enabled by VDU 2
1	Unused
2	Paged scrolling selected by VDU 14
3	Text window in force (ie software scrolling)
4	In a shadow mode
5	In VDU 5 mode
6	Cursor editing in progress
7	Screen disabled with VDU 21

OS_Byte &86 (134) – Read text cursor position

On entry: –

On exit: R1 = horizontal position
R2 = vertical position

This call returns the current text cursor position unless cursor editing is in progress, in which case the position returned is that of the input cursor. OS_Byte &A5 (165) reads the position of the output cursor irrespective of cursor editing mode.

Text is printed at horizontal positions 0 to n-1, where 'n' is the number of characters per line in the current text window. Therefore, the value obtained is normally in this range. However, if there is a pending newline (see VDU 23,16), a position of 'n' will be returned.

OS_Byte &A5 (165) – Read output cursor position

On entry: –

On exit: R1 = horizontal position
R2 = vertical position

This call returns the position of the output cursor, even while cursor editing is in progress.

OS_Byte &87 (135) – Read character at text cursor position and screen mode

On entry: –

On exit: R1 = ASCII code of character (0 if unreadable)
R2 = screen mode

This call returns the screen mode and the ASCII code of the character at the text cursor position. If cursor editing is in progress, it returns the character code returned by the character at the input cursor position (ie the character that would be copied the next time **Copy** is pressed).

Note that the screen mode does not have bit 7 set, even if it is a shadow mode.

OS_Byte &90 (144) – Set vertical screen shift and interlace

On entry: R1 = vertical screen shift
R2 = interlace flag

On exit: R1 = previous screen shift
R2 = previous interlace flag

This call specifies the vertical screen alignment and interlace options after the next mode change. R1 sets the vertical offset. R2 turns interlace on and off as follows:

Value	Meaning
0	Interlace on
1	Interlace off

It is equivalent to OS command *TV, to which you should refer for details of the arguments.

OS_Byte &A0 (160) – Read VDU variable value

On entry: R1 = VDU variable number (0 – 15)

On exit: R1 = value of the variable
R2 = value of next variable (which would be read with R1+1)

The VDU driver uses a number of locations in RAM to store transient information. This call allows some of these locations to be examined. Note that the variables are not necessarily stored in the order implied by the value of R1 on entry. However, the relationship between R1 and the variable read is guaranteed to remain the same for all versions of the OS.

Call	Location
R1 = 0	LSB of graphics window left column (ic)
R1 = 1	MSB of graphics window left column (ic)
R1 = 2	LSB of graphics window bottom row (ic)
R1 = 3	MSB of graphics window bottom row (ic)
R1 = 4	LSB of graphics window right column (ic)
R1 = 5	MSB of graphics window right column (ic)
R1 = 6	LSB of graphics window top row (ic)
R1 = 7	MSB of graphics window top row (ic)
R1 = 8	Text window left column
R1 = 9	Text window bottom row
R1 = 10	Text window right column
R1 = 11	Text window top row
R1 = 12	LSB of graphics origin X co-ordinate (ec)
R1 = 13	MSB of graphics origin X co-ordinate (ec)
R1 = 14	LSB of graphics origin Y co-ordinate (ec)
R1 = 15	MSB of graphics origin Y co-ordinate (ec)

– *Note:* (ic) means internal co-ordinates: the origin is always the bottom left of the screen. One unit is one pixel wide and one pixel high.

(ec) means external co-ordinates: the screen is 1280 units wide by 1024 units high in all modes except 22, where it is 976 units high.

This OS_Byte is provided mainly for compatibility with the BBC/Master 128 OS. You can read many more of the VDU variables using OS_ReadVduVariables &31 (49) and OS_ReadModeVariable &35 (53).

OS_Byte &A3 (163) – Read / write general graphics information

On entry: R1 = 242
R2 = dot-dash repeat length or action code

On exit: R1 is preserved or contains status information
R2 is preserved or contains status information

This call is a general purpose one reserved for Acorn applications. The only value of R1 which is guaranteed to perform a useful function is 242. The type of action depends on the value of R2:

Value	Meaning
0	Set default dot-dash pattern and length
1 – 64	Set dot-dash line repeat length to the value given
65	Return status information
66	Return information on the current sprite

The status information is returned in R1 and R2 as follows:

R1 bits	Meaning
Bit 7 = 1	(Sprites are always active)
Bit 6 = 1	(Flood fill is always active)
Bits 0 – 5	Current dot dash line repeat length

R2 bits	Meaning
Bits 0 – 31	Size of sprite workspace in bytes This is 8K/32K times the configuration value SpriteSize

The information on the current sprite is returned in R1 and R2 as follows:

R1 = width in pixels (ie internal co-ordinates)
R2 = height in pixels (ie internal co-ordinates)

OS_Byte &C1 (193) – Read/write flash counter

On entry: R1 = 0
R2 = 255

On exit: R1 = previous value
R2 = value of next location (duration of second colour)

This call accesses the location used as a count-down timer for the flashing colours. The location is loaded with the count for the first colour and decremented at a VSYNC rate. When it reaches zero, the colours are swapped and the counter is loaded with the duration of the second colour.

OS_Byte &D3 (211) – Read/write bell channel

On entry: R1 = 0 or new channel
R2 = 255 or 0


On exit: R1 = previous channel
R2 = value of next location (bell sound information)

The bell (VDU 7) sound is output on channel 1 by default. This call provides a means of determining the current channel or changing it if required.

OS_Byte &D4 (212) – Read/write bell sound volume

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location (bell frequency)

This allows you to read or set the volume of the sound used to make the  bell sound. Values for the amplitude are in the range &80 (loudest) to &F8 (softest) in steps of &08. The default setting depends on the *Configure Loud/Quiet setting (&90/&D0 respectively).

OS_Byte &D5 (213) – Read/write bell frequency

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location (bell duration)

This call provides a means of reading or changing the frequency associated with the bell sound. The default value is 100, and it has the same interpretation as the third parameter of the SOUND statement in BASIC and the *SOUND command.

OS_Byte &D6 (214) – Read/write bell duration

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 is undefined

This call provides a means of reading or changing the duration of the bell sound. The default value is 6, and the unit is 20ths of a second.

OS_Byte &D9 (217) – Read/write paged mode line count

On entry: R1 = 0 or new count
R2 = 255 or 0

On exit: R1 = previous count
R2 = value of next location (bytes in VDU queue)

In the paged output mode, the display is prevented from scrolling (awaiting the depression of **Shift**) when approximately 75% of the height of the current text window has been scrolled. The number of lines printed since the last page halt is maintained in the location accessed by this call and it may be either read or changed (normally to 0 before requesting user input).

If you are using OS_Word &00 or OS_ReadLine to perform the input, this call is made automatically.

OS_Byte &DA (218) – Read/write bytes in VDU queue

On entry: R1 = 0 or new count
R2 = 255 or 0

On exit: R1 = previous count
R2 = value of next location (TAB key code)

This call affects the count of the number of characters which remain to be passed to the VDU driver in order to complete the current VDU sequence. The value is (minus the number of bytes left), and is held in 2's complement notation (eg &FF means one byte to go). The call may be used to read the value or to change it (normally to zero, which has the effect of abandoning an incomplete VDU command).

When an escape condition is acknowledged, this call is made automatically. This prevents the first few characters of an error message from being 'swallowed' by an incomplete VDU sequence.

THE VDU OS_WORDS

This section describes the OS_Word calls which affect the VDU drivers. Many of these are status-returning calls, rather than being routines which affect the screen. They are listed in numerical order of R0 on entry.

OS_Word &09 (9) – Read pixel logical colour

Parameter block size: 5

On entry: The first four bytes of the parameter block contain the co-ordinate of the pixel:

R1+0 = LSB of X co-ordinate
R1+1 = MSB of X co-ordinate
R1+2 = LSB of Y co-ordinate
R1+3 = MSB of Y co-ordinate

On exit: R1+4 = the logical colour of the pixel specified.

This call determines the logical colour of the pixel at given co-ordinates on the graphics screen. If the colour is returned as &FF then either:

– the screen is in a 256 colour mode and the logical colour was 255

- the pixel is off the screen
- the screen is in a non-graphics mode.

To overcome the ambiguity caused by 256 colour modes, you should use `OS_ReadPoint &32 (50)` instead. This returns both the logical colour and tint. The `OS_Word` should be used for compatibility purposes only.

`OS_Word &0A (10) – Read a character definition`

Parameter block size: 9

On entry: The first byte of the parameter block contains the ASCII code of the character required:

`R1+0 = ASCII code of character required`

On exit: Bytes `R1+1` to `R1+8` contain the definition of the specified character:

`R1+1 = top row of definition`

. .
. .

`R1+8 = bottom row of definition`

The characters displayed in all modes other than teletext mode are defined as an eight-by-eight matrix of dots. This call enables you to read the definition for a specified ASCII code. However, the definitions returned for ASCII codes 0 to 31 and 127 (ie the non-printing characters) are not meaningful.

Bits set in each row of the character definition are displayed in the current text foreground colour; bits clear in each row are displayed in the current text background colour. In VDU 5 mode, bits which are set are plotted in the graphics foreground colour and action; bits which are clear are not plotted at all.

OS_Word &0B (11) – Read the palette

Parameter block size: 5

On entry: The first byte of the parameter block contains the logical colour:

R1+0 = logical colour to read

On exit: Details of the physical colour are returned in the last four bytes:

R1+1 = physical colour associated with the specified logical colour

R1+2 = red component

R1+3 = green component

R1+4 = blue component

This call allows you to determine the physical colour associated with a particular logical colour. The call can only return one of the colours associated with a flashing colour. To read the full information about a logical colour's palette entry, you should use OS_ReadPalette &2F (47). The OS_Word is provided for compatibility only.

OS_Word &0C (12) – Write the palette

Parameter block size: 5

On entry: The parameter block contains details of the logical colour and the new physical colour which is to be associated with it:

R1+0 = logical colour to change

R1+1 = new physical colour

R1+2 = red component

R1+3 = green component

R1+4 = blue component

On exit: The parameter block remains unchanged.

This call allows you to change the physical colour associated with a particular logical colour. It duplicates the function of VDU 19 command. However, the OS_Word

call is faster and may be used in interrupt routines. The five bytes of the parameter block are equivalent to the five parameters l,p,r,g,b described in the section on VDU 19.

OS_Word &0D (13) – Read current and previous graphics cursor positions

Parameter block size: 8

On entry: The parameter block is unused.

On exit: The X and Y co-ordinates are returned in the parameter block:

R1+0 = LSB of previous X co-ordinate
 R1+1 = MSB of previous X co-ordinate
 R1+2 = LSB of previous Y co-ordinate
 R1+3 = MSB of previous Y co-ordinate
 R1+4 = LSB of current X co-ordinate
 R1+5 = MSB of current X co-ordinate
 R1+6 = LSB of current Y co-ordinate
 R1+7 = MSB of current Y co-ordinate

All the co-ordinates are in external form. You can read points visited before the previous one (and many other VDU variables) using OS_ReadVduVariables &31 (49).

OS_Word &16 (22) – Write screen base address

Parameter block size: 5

On entry: The parameter block contains the following values:

R1+0 = Type
 R1+1 = Least significant byte of offset
 R1+2 ...
 R1+3 ...
 R1+4 = Most significant byte of offset

On exit: The parameter block remains unaltered.

This routine sets up a new screen base address. It is given as the offset from the address of the base of the screen buffer to the start of the screen display. This address can be used as the area of the buffer which is to be updated, ie written to by the VDU drivers, or the area which is to be displayed by the hardware, or both, depending on the bits of the first byte in the parameter block:

Bit 0	Used by VDU drivers
Bit 1	Displayed by hardware

This allows multiple screens to be used. For example, in mode 12 two copies of the screen can be kept. One of these can be updated whilst the other is being displayed using the following parameter blocks:

R1+0	Contains 2	Displayed
R1+1 – R1+4	Contains &00	
R1+0	Contains 1	Updated
R1+1 – R1+4	Contains &14000	

Then the two screens can be swapped over (at VSYNC) by changing over the addresses so that smooth animation is obtained.

The size of the screen buffer is given by the *CONFIGURE ScreenSize parameter. Thus if this is set to 10 on a 300 series machine, the buffer size is 10*8192 or 81,920 bytes. This allows for four 20K mode screens, two 40K modes, or one 80K mode screen. See the above *CONFIGURE command for more details.

A slightly simpler way of achieving bank switching is to use OS_Bytes &70 – &71 (112 – 113). With these, you only have to specify the bank number, not the actual offset.

THE VDU SWI CALLS

This section describes the OS SWI calls (except OS_Bytes and OS_Words) which affect the VDU drivers. Many of these are status-returning calls, rather than routines which affect the screen. They are listed in numerical order of SWI number.

OS_ReadPalette &2F (47)

On entry: R0 = logical colour
R1 = which colour

On exit: R2 = setting of first flashing colour
R3 = setting of second flashing colour

OS_ReadPalette reads the setting of a particular colour. R1 selects whether the normal colour, border colour or cursor colour is read as follows:

Value	Meaning
16	Read normal colour
24	Read border colour
25	Read cursor colour

The settings for the first flash colour and second flash colour are returned in R2 and R3 respectively. If these are identical then the colour is a steady, non-flashing one. The value contained in each of these is interpreted as follows:

Bits	Meaning
0 - 7	Value showing how colour was programmed
8 - 15	Amount of red
16 - 23	Amount of green
24 - 31	Amount of blue

The bottom byte (bits 0 - 7) returns the value of the second parameter to the VDU 19 command which defines the palette. For example:

Value	Meaning
0 – 15	Actual colour (BBC compatible)
16	Defined by giving amounts of red, green and blue
17 – 18	Flashing colour defined by giving amounts of red, green and blue

OS_ReadVduVariables &31 (49)

On entry: R0 = pointer to the input block
R1 = pointer to the output block

On exit: –

OS_ReadVduVariables reads in a series of VDU variables and places them in sequence into a block of memory. The input block consists of a sequence of words. Each word is the number of the variable to be read. A value of -1 terminates the list. The value of each variable is put as a word into the output block, any invalid variables being entered as zero. The output block has no terminator. Both blocks must be word-aligned.

The possible variable numbers are the same as for OS_ReadModeVariable (see below) with the following additions:

GWLCol	128	Left-hand column of the graphics window (ic)
GWBRow	129	Bottom row of the graphics window (ic)
GWRCol	130	Right-hand column of the graphics window (ic)
GWTRow	131	Top row of the graphics window (ic)
TWLCol	132	Left-hand column of the text window
TWBRow	133	Bottom row of the text window
TWRCol	134	Right-hand column of the text window
TWTRow	135	Top row of the text window
OrgX	136	X co-ordinate of the graphics origin (ec)
OrgY	137	Y co-ordinate of the graphics origin (ec)
GCsX	138	X co-ordinate of the graphics cursor (ec)
GCsY	139	Y co-ordinate of the graphics cursor (ec)
OlderCsX	140	X co-ordinate of oldest graphics cursor (ic)
OlderCsY	141	Y co-ordinate of oldest graphics cursor (ic)

OldCsX	142	X co-ordinate of previous graphics cursor (ic)
OldCsY	143	Y co-ordinate of previous graphics cursor (ic)
GCsIX	144	X co-ordinate of graphics cursor (ic)
GCsIY	145	Y co-ordinate of graphics cursor (ic)
NewPtX	146	X co-ordinate of new point (ic)
NewPtY	147	Y co-ordinate of new point (ic)
ScreenStart	148	Address of the start of screen used by VDU drivers
DisplayStart	149	Address of the start of screen used by display hardware
TotalScreenSize	150	Size of configured screen memory
GPLFMD	151	GCOL action for foreground colour
GPLBMD	152	GCOL action for background colour
GFCOL	153	Graphics foreground colour
GBCOL	154	Graphics background colour
TForeCol	155	Text foreground colour
TBackCol	156	Text background colour
GFTint	157	Tint for graphics foreground colour
GBTint	158	Tint for graphics background colour
TFTint	159	Tint for text foreground colour
TBTint	160	Tint for text background colour
MaxMode	161	Highest mode number available

– *Note:* (ic) means internal co-ordinates, where (0,0) is always the bottom left of the screen. One unit is one pixel.

(ec) means external co-ordinates, where (0,0) means the graphics origin, and the size of one unit depends on the resolution. The screen is always 1280 by 1024 external units. The graphics origin is stored in external co-ordinate units, but is relative to the bottom left of the screen.

The 'new point' is the internal form of the co-ordinates given in an unrecognised PLOT command. When the UKPlot vector is called, the internal format co-ordinates (variables 140 – 145) have not yet been shuffled down, so the graphics cursor (144 – 5) contains the co-ordinates of the last point visited. The external co-ordinates version of the current point (138 – 9) is updated from the co-ordinate given in the unrecognised plot.

OS_ReadPoint &32 (50)

On entry: R0 = x co-ordinate
R1 = y co-ordinate

On exit: R2 = colour
R3 = tint
R4 = screen flag

The co-ordinates are in external units and are relative to the current graphics origin.

OS_ReadPoint takes a point and returns its colour in R2 and its tint setting (amount of white, in the range 0 – 255) in R3. R4 returns the following:

Value	Meaning
0	Point on the screen
-1	Point off the screen (R2 = -1 also)

OS_ReadModeVariable &35 (53)

On entry: R0 = screen mode
R1 = variable number

On exit: R2 = value of variable
C is set if variable or mode numbers were invalid

OS_ReadModeVariable allows you to read information about a particular screen mode without having to change into that mode. The possible variable numbers are given below:

ModeFlags	0	The bits of the result have the following meanings:
		Bit 0 = 0 graphics mode = 1 non-graphics mode
		Bit 1 = 0 non-teletext mode = 1 teletext mode
		Bit 2 = 0 non-gap mode = 1 gap mode
ScrRCol	1	Maximum column number for printing text ie number of columns - 1
ScrBCol	2	Maximum row number for printing text ie number of rows - 1
NColour	3	Maximum logical colour ie either 1, 3, 15 or 63 (not 255)
XEigFactor	4	This indicates the horizontal pixel resolution:
		0 1280 pixels
		1 640 pixels
		2 320 pixels
		3 160 pixels

For values 1, 2 and 3 it is the number of bits by which 1280 must be shifted right in order to obtain the number of screen pixels. Note that it gives the physical pixel size, which isn't necessarily the same as the number of pixels that appear to be available. For example, in mode 4 the value is 1, implying 640 pixels. This is because in terms of the hardware mode, mode 4 is really mode 0, with each pixel replicated horizontally during drawing.

YEigFactor	5	This indicates the vertical pixel resolution:
		1 512 pixels
		2 256 pixels

Again, this is the number of bits by which 1024 must be shifted right to obtain the actual number of vertical pixels.

LineLength	6	Number of bytes on a pixel row
------------	---	--------------------------------

This is the same as (characters per row)*(bits per pixel). For example, in mode 15 it is 80*8, or 640.

ScreenSize	7	Number of bytes one screen buffer occupies
------------	---	--

YShftFactor	8	Scaling factor for start address of a screen row
-------------	---	--

The value is LOG base 2 of (LineLength)/5. Thus to work out the address of the first byte of a given screen row, y, you perform (y<<YShftFactor)*5. This gives the offset from the top of the screen, so this must be added to it to find the actual address.

Log2BPP	9	LOG base 2 of the number of bits per pixel
Log2BPC	10	LOG base 2 of the number of bytes per character

It is in fact the LOG base 2 of the number of bytes per character divided by eight. So in mode 0, for example, it is LOG base 2 of (8/8), or 0. In mode 15 it is LOG base 2 of (64/8), or 3. It would be exactly the same as Log2BPP, except for the 'double pixel' modes.

OS_RemoveCursors &36 (54)

On entry: -

On exit: -

OS_RemoveCursors removes the cursors (output and copy, if active) from the screen, saving the old state (their positions, flash rate etc.) on an internal stack so

that it may be recovered later. This instruction must always be balanced later by a `OS_RestoreCursors` to restore the cursor again.

`OS_RestoreCursors &37 (55)`

On entry: –

On exit: –

`OS_RestoreCursors` restores the cursor state previously saved on the internal stack using `OS_RemoveCursors`.

`OS_CheckModeValid &3F (63)`

On entry: R0 = mode number to check

On exit: C = 0 if mode is valid, 1 otherwise
 If C=0, R0 is preserved, otherwise:
 R0 = -1 if the mode is non-existent
 R0 = -2 if there is not enough memory
 R1 = mode that will be used

`OS_CheckModeValid` determines whether you can change to a given mode and return with the appropriate carry set. R0 = -1 on exit implies that the mode you are checking isn't available on the current type of monitor. R1 contains the mode that will be used if an attempt is made to select the mode which you are checking, using VDU 22.

`OS_ClaimScreenMemory &41 (65)`

This call allows the screen memory to be used as a temporary buffer area. See the chapter **MEMORY MANAGEMENT** for details.

THE MOUSE AND POINTER

The mouse pointer is a hardware sprite or 'cursor' generated by the VIDC video controller chip. It differs from the sprites plotted by the Sprite module. These are software entities which are displayed by plotting pixels on the screen in the appropriate colours. The mouse pointer has its own screen RAM, and is superimposed on the main screen.

To use the mouse pointer, you must define its shape using the appropriate OS_Word call. It may be up to 32 pixels square. The size of the pointer pixels is the same as those in current display mode. However, the number of colours available is always four.

The shape is stored with two bits per pixel, so pointer colours are in the range 0–3. Colour 0 is always transparent. Pixels set to this colour are not displayed, so the main screen display can be seen underneath. The other three colours have their own palette entries, so may be programmed separately from the other colours on the screen.

OS_Byte and OS_Word calls control the pointer shape. Because the pointer is so often used in conjunction with the mouse, you can tie the two together, so that the pointer follows the mouse. In addition, you can set 'scaling factors', which determine how much the pointer moves for a given movement of the mouse.

Because of this close association between the pointer and the mouse, this section also describes calls relating to the mouse itself. However, it doesn't cover the mouse/pointer support calls provided by the window manager module. These calls should be used in programs running under the Wimp environment, in preference to those documented here. See the chapter **THE WINDOW MANAGER** for details.

Mouse/pointer OS_Byte calls

OS_Byte &6A (106) – Select pointer / activate mouse

On entry: R1 = pointer shape and linkage flag (see below)

On exit: R1 = old pointer shape number and linkage state
R2 is undefined

You can define four 'pointer buffers' using OS_Word &15 (21) (see below), each holding a different shape definition for the mouse pointer. The present call allows you to select one of these definitions for future use, or to turn off the pointer depending on the bottom three bits of R1:

Value	Meaning
0	Turn off current pointer
1 - 4	Select given pointer

If a pointer is selected it can be linked to the mouse so the mouse drives it, depending on bit seven of R1 as follows:

Value	Meaning
0	Link pointer to mouse
1	Pointer unlinked

For example, a value in R1 of &03 selects pointer three and links it to the mouse, and a value of &82 selects pointer two but leaves it unlinked.

OS_Byte &80 (128) – Get buffer/mouse status

On entry: R1 = action code

On exit: R1 = low byte of position or number of bytes in buffer/free
R2 = high byte of position or number of bytes in buffer/free

The action of this call depends upon the value in R1. It determines the current x or y position of the mouse or the number of bytes in a particular input buffer or the number of free bytes in a particular output buffer. In particular, if R0=246 on entry, the call returns the number of bytes in the mouse buffer.

On entry **On exit**

R1 = 7 R1 = x position MOD 256; R2 = x position DIV 256
R1 = 8 R1 = y position MOD 256; R2 = y position DIV 256
R1 = 246 R1 & R2 contain the number of bytes in mouse buffer

The mouse buffer is 63 bytes long. An entry is inserted into it every time one of the buttons on the mouse changes state. Each entry is nine bytes long. It consists of two two-byte co-ordinates, a button state byte, and a four-byte time stamp. Up to seven button changes may therefore be buffered at once.

It is not recommended that you use this OS_Byte to read the mouse co-ordinates, unless you require only one co-ordinate. This is because two calls are required to discover both x and y, and the mouse might change position between the two calls. The OS_Mouse call is more useful, as it returns both the mouse button state and the time-stamp. (See below.)

Mouse/pointer OS_Word call

OS_Word &15 (21) – Define pointer and mouse parameters

This provides four different functions associated with the pointer and the mouse.

Define pointer size, shape and active point

Parameter block size: 10

On entry: The parameter block contains all the information:

R1+0 = 0
R1+1 = Shape number (1 – 4)
R1+2 = Width (w) in bytes (0 – 8)
R1+3 = Height (h) in pixels (0 – 32)
R1+4 = ActiveX in pixels from left (0 – w*4–1)
R1+5 = ActiveY in pixels from top (0 – h–1)

R1+6 = Least significant byte of pointer (P) to data
 R1+7 .
 R1+8 .
 R1+9 = Most significant byte of pointer to data

On exit: The parameter block remains unchanged.

You can define four shapes. These are numbered one to four and may be selected using OS_Byte &6A (106).

As the pointer is always displayed in 2 bits per pixel (four pixels per byte), and the maximum width in bytes is 8, the maximum width is 32 pixels.

The ActiveX and ActiveY entries give the distance of the cursor 'hot spot' from the top left corner of the pointer. If these are zero, then positioning the pointer at co-ordinates (x,y) will move the top left corner to that position. Suppose the shape was a cross-hair 9 pixels in each direction; then making ActiveX and ActiveY (5,5) would position the hot-spot at the centre of the cross.

The data for the shape is pointed to by R1+6 – R1+9. This data table contains the information for each row, from top to bottom, and the data within each row is given from left to right. Each byte contains the colours for four pixels. Bits 0,1 hold the colour number for the left-most pixel, bits 6,7 the colour for the right-most pixel. (So the pixels are displayed in reverse order to the order in which the byte would be written down.)

Colour zero is always transparent (ie the screen information shows through pixels in this colour). The other three colours may be set independently of any other colours on the screen using VDU 19 or the equivalent OS_Word.

Define mouse co-ordinate bounding box

Parameter block size: 9

On entry: The parameter block contains the new co-ordinates of the box:

R1+0 = 1
R1+1 = LSB of left co-ordinate all treated as
R1+2 = MSB of left co-ordinate signed 16-bit values,
R1+3 = LSB of bottom co-ordinate relative to screen origin at the time
R1+4 = MSB of bottom co-ordinate the command is issued
R1+5 = LSB of right co-ordinate
R1+6 = MSB of right co-ordinate
R1+7 = LSB of top co-ordinate
R1+8 = MSB of top co-ordinate

On exit: The parameter block remains unchanged.

The co-ordinates should be given as signed 16-bit values relative to the screen origin at the time the command is issued.

If (left > right) or (bottom > top) then the command is ignored.

An infinite box can be obtained by setting:

left = &8000 (-32768)
bottom = &8000 (-32768)
right = &7FFF (32767)
top = &7FFF (32767)

If the current mouse position is outside the box, it is homed to the nearest point inside the box.

The default box is (0,0) to (1279,1023), ie the same as the default graphics window.

Define mouse multipliers

Parameter block size: 3

On entry: The parameter block contains the X and Y multipliers:

R1+0 = 2
 R1+1 = X multiplier
 R1+2 = Y multiplier

(both treated as signed eight-bit values)

The multipliers control the ratio between the movement of the mouse and the change in the co-ordinates of the mouse. The higher each value, the greater the amount the pointer moves (if linked to the mouse) for a given movement of the mouse.

The multipliers should both be given as signed eight-bit values. By specifying negative values (eg 255 for -1), you can make the point move in the opposite direction from usual.

Both multipliers default to 1. With this setting, a movement of approximately 15cm of the mouse will move the pointer across the screen (1280 units).

Set mouse position

Parameter block size: 5

On entry: The parameter block contains the new X and Y positions:

R1+0 = 3
 R1+1 = LSB of X position
 R1+2 = MSB of X position
 R1+3 = LSB of Y position
 R1+4 = MSB of Y position

The new values for the X and Y positions of the mouse are given as two signed 16-bit values. If the new position lies outside the bounding box of the mouse, this command will be ignored.

Note that this call sets the position of the mouse rather than the pointer. If the mouse and pointer are not linked, the position of the pointer on the screen is left unchanged.

Mouse/pointer SWI call

You can use the call `OS_Mouse` to discover the state of the mouse: its co-ordinates (and therefore the position of the pointer if it is active and linked), the state of the mouse buttons, and a 'time-stamp' for the reading.

`OS_Mouse` &1C (28)

On entry: --

On exit: R0 = mouse X co-ordinate
R1 = mouse Y co-ordinate
R2 = mouse buttons
R3 = time of button change

`OS_Mouse` reads from the mouse buffer the mouse X and Y positions as values between -32768 and 32767 . Unless the graphics origin has been changed, the range will usually be $0 - 1279$ for x and $0 - 1023$ for y. The call also returns buttons currently pressed as a value in the range $0 - 7$:

Bit	Meaning when set
0	Right button down
1	Middle button down
2	Left button down

If there is no entry in the mouse buffer, the current status is returned. R3 gives the time the entry was buffered, or the current time if it is not a buffered reading. It uses the monotonic timer (see `OS_ReadMonotonicTime`).

THE VDU EXTENSION VECTOR

This section assumes you are familiar with vectors.

It is possible to replace totally the VDU driver software. This can be useful if you want to change the characteristics of screen output in a dramatic way, eg as the Font manager does. When you call `OS_WriteC`, the OS checks bit 1 of the stream's byte

(see OS_Byte &03). If this is clear, the VDU is enabled. If it is set, the VDU stream is disabled, and no attempt is made to use the VDU driver.

If the VDU is enabled, the OS then checks bit 5 of the stream's byte. If this is clear, the default VDU drivers are called; this is the usual action. If bit 5 is set, the OS instead calls the code which is vectored through the vector VDUXV (number &1B). If no one is using VDUXV, the default action is to do nothing.

When VDUXV is called, the character to be printed is in R0, as on entry to OS_WriteC. On exit, this and all other registers should be preserved. The action taken by a routine using VDUXV is entirely dependent on that routine. Note that if the routine needs to continue to interpret certain multi-byte sequences, it must maintain its own VDU queue. It should still use OS_Byte &DA (218) to maintain the VDU queue pointer, so that sequences can still be abandoned in the usual way.

Usually, the routine would claim the call, rather than pass it on. It is unlikely that useful results would be obtained by two or more routines intercepting VDUXV. This means, of course, that only the most recent routine to OS_Claim VDUXV will get to hear about it.

If, on return, the carry flag is set, then the OS will attempt to send the character to the current printer. Whether the character is actually printed depends on bits 2 and 6 of the stream's byte, and on the printer ignore status.

A typical return sequence for a routine intercepting VDUXV which wants the character to be printed too would be:

```
LDM    R13!,R14          ;Get return address from the stack
ORRS   PC,R14,#carry     ;Set the carry flag and return
```

where carry is $2 \ll 25$, ie the carry bit in R15.

CHARACTER INPUT

This chapter describes the ways in which characters may be read into the Archimedes. Just as there are several output streams, so there is more than one place from which characters may be read. Although there are fewer input streams than output streams, there are more ways of reading characters than writing them. In particular, you can deal with the keyboard, the principal input stream for many applications, at a variety of levels.

THE INPUT STREAMS

There are two and a half input streams. That is, there are two selectable streams, the keyboard and the RS423 port, and one other, which may override either of these if it is active. This last is the *EXEC file, which performs a similar job for input to the one performed by the *SPOOL file for output.

Input streams are mutually exclusive: whereas a character sent to OS_WriteC finds its way to all of the currently enabled outputs, a request to read a character can only ever come from a single input source.

Both the keyboard and the RS423 port are buffered devices. The keyboard input buffer can hold 31 characters. It is often termed a type-ahead buffer, as it enables the user to type in commands ahead of the input requests which will read those commands.

The RS423 input buffer holds 255 bytes. Associated with it is a 'handshake extent'. This is a count of the number of spaces which must be left in the buffer before the RS423 software in the OS asks the device attached to the RS423 port to stop sending. This ensures that characters are never lost as a result of software being unable to keep up with incoming characters.

A number of events are associated with these input devices. In particular:

- input buffer full
- character entering keyboard buffer
- key press/release
- RS423 error
- escape pressed events.

See the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS** for further details. Also, as with all buffers, you can insert characters, remove characters, examine the buffer etc. See the section **Buffers** in the same chapter for further details.

To select one of the two input streams, `OS_Byte &02` is used:

`OS_Byte &02 (2)` – Specify input stream

On entry: R1 = action code (0, 1 or 2)

On exit: R1 = previous device (0 or 1)
R2 is undefined

This call selects the device from which all subsequent input is taken. This is determined by R1 as shown below:

Value	Device
0	Keyboard input and disables the RS423 port
1	RS423 input
2	Keyboard input and enables the RS423 port

The previous input device is returned in R1 as follows:

0	Input was from the keyboard
1	Input was from the RS423 port

The difference between using R1=0 and R1=2 is that the latter enables characters to be received into the RS423 input buffer under interrupts, even though characters will be read from the keyboard input buffer. If the input stream is subsequently switched to the RS423, using R1=1 then those characters received under interrupts will be read.

The input device may be read (but not set) using the `OS_Byte &B1 (177)`. See below.

OS_Byte &B1 (177) – Read input source

On entry: R1 = 0
R2 = 255

On exit: R1 = previous buffer number
R2 = value of the next location (keyboard semaphore)

BASIC INPUT ROUTINES

This section describes the fundamental routines used to read characters from the current input stream. For many purposes, these routines are all that is required to interact with the user. However, some applications may require a lower-level interaction with the input devices, so the next two sections provide more detailed information about the keyboard and RS423 input streams respectively.

To read a single character, you use either OS_ReadC or OS_Byte &81 (129). The difference is that the former will always wait until a character is available (or an escape condition arises), whereas the latter will time-out after a predetermined number of centi-seconds.

OS_ReadC &04 – Read Character

On entry: –

On exit: R0 = ASCII code or error type
C = 0 if R0 is a valid character
C = 1 if R0 is an error type – R0 = &1B means escape

OS_Byte &81 (129) – Read key with time limit

On entry: R1 = time limit low byte
R2 = time limit high byte

On exit: R1 = ASCII code, or &FF if R1 = &FF (timeout)
R2 = &00 if character read
R2 = &1B if **[Escape]** pressed
R2 = &FF if timeout

This call reads a key from the keyboard subject to a specified time limit or performs a keyboard scan for a specified key depression. The second use is covered in the section below on the keyboard.

To read a key within a specified time limit, use R1 and R2 to indicate the time limit (n) in hundredths of a second as follows:

R1 = n MOD &100
R2 = n DIV &100

The upper limit is 32767 centi-seconds. Information about the first key pressed (if one was pressed within the time limit given) is returned in R1 and R2.

These two routines are equivalent to the BASIC functions GET and INKEY respectively. If, having used one of them, you detect an escape condition, you should acknowledge it or clear it using one of the OS_Bytes described in the section **The escape condition**.

While the OS is waiting for a character to appear in the appropriate input buffer during one of these calls, it also deals with cursor key presses. That is, if one of the arrow keys is pressed, cursor edit mode is entered, indicated by the presence of two cursors on the screen. You can copy characters from underneath the input cursor by pressing **[Copy]**. The character read is returned from the routine as if you had typed it explicitly. Cursor editing is cancelled when ASCII 13 is sent to the VDU driver.

LINE INPUT

A routine is provided to ask for a whole line of input 'in one go'. The line is terminated when you press **[Enter]**, **[Ctrl]**, or the current escape key. The routine is described below.

OS_ReadLine &OE (14) – Read line from input stream to memory

On entry: R0 = pointer to the buffer to hold text
 R1 = maximum possible length of line
 R2 = lowest character which will be placed in buffer
 R3 = highest character to be placed in buffer

On exit: R1 = length of buffer (not including carriage return)
 C is set if input was terminated by an escape

- OS_ReadLine reads a line of text from the current input stream
- R0 points to the buffer where the text is to be placed
- R1 contains the maximum possible length of the line
- R2 contains the lowest character code which is to be placed in the buffer (excluding carriage return)
- R3 contains the highest character code which is to be placed in the buffer.

On exit, R1 contains the number of characters in the buffer, not counting the carriage return. C is set if the input was terminated by the escape character, and no return is stored in the buffer.

If the input is not terminated by an escape, then the last character in the buffer is always ASCII 13 (carriage return), even if you press **[Ctrl]J** to terminate the input.

All input characters (with the exception of ASCII 21 or **[Ctrl]U**) are sent to the output stream currently selected; only characters within the range specified by R2 and R3 are actually put into the buffer.

If you press **[Delete]** (ASCII 127) and there are characters in the buffer, the **[Delete]** is echoed on the screen and the last character entered into the buffer is removed. If the buffer is empty, **[Delete]** is ignored and not output.

CtrlU (ASCII 21) deletes all the characters placed in the buffer and sends an ASCII 127 to the output stream for each character that was in the buffer, effectively erasing the line from the screen.

If the count of the number of characters input reaches the number specified by R1, further characters are ignored and cause ASCII 7 (**Ctrl**G) to be sent to the output stream, which will usually cause a short beep to be emitted. Note that **Delete** and **Ctrl**U will still function as described above.

OS_ReadLine must not be used in an interrupt or event routine.

Note that OS_ReadLine uses OS_ReadC when reading individual characters, so it can be used to read lines from the RS423 stream or the *EXEC file.

OS_Word &00 is equivalent to this routine, and is provided for compatibility with old programs. It is documented here for completeness.

OS_Word &00 (0) – Read line from input stream to memory

Parameter block size: 5

On entry: The parameter block contains the details of the buffer into which characters are read, the maximum number of characters to be read, and limiting ASCII codes as follows:

R1+0 = LSB of buffer address	OS_ReadLine R0
R1+1 = MSB of buffer address	
R1+2 = maximum number of characters	OS_ReadLine R1
R1+3 = lowest ASCII code	OS_ReadLine R2
R1+4 = highest ASCII code	OS_ReadLine R3

On exit: The parameter block is unaltered. R2 and the carry flag are set up as follows:

C = 0	Line of input was terminated normally
C = 1	Line of input was terminated by an escape condition
R2 =	length of input line (not including the carriage return)

The buffer must be in the bottom 64K of memory.

THE KEYBOARD

This section describes in more detail how the keyboard is handled by the Archimedes. It also explains how characters which enter the keyboard buffer are interpreted once they come to be read by one of the routines documented in the previous sections.

Keyboard interrupts

When a key is pressed (or released), a code unique to that key is transmitted to the Archimedes through the keyboard connector cable. This code is read into a chip called the I/O controller or IOC, which then causes an interrupt to occur. The OS responds to this interrupt by reading the keycode from the IOC, and passing it on to the keyboard handler for further processing.

At this stage, a key press/release event may be generated, which you can handle as required. Also, at this level mouse button presses look exactly the same as any other key press. They are, however, treated separately by the OS. The OS doesn't pass on mouse button presses to the keyboard handler, although they can be made to generate an event.

It is possible to customise totally the way in which the keyboard works by writing your own keyboard handler. However, this is outside of the scope of this discussion, as this chapter concentrates on the actions of the built-in key handler.

The first thing the keyboard handler must do is convert the keycode into a form which is more like the ASCII codes the user expects. The keycode sent to the IOC is a number which bears a simple relationship to the position of the key on the keyboard. These numbers are listed in a table in the section on key press/release events. They bear no relationship to the legends on the keyboard.

If the key pressed (or released) is one of the shifting keys, **Shift**, **Ctrl**, **Alt**, or one of the locking keys **Caps Lock**, **Num Lock** or **Scroll Lock** is pressed, then the key handler just makes a note of this fact by updating its status information; it doesn't cause any character to be inserted into the keyboard buffer.

If the key pressed was one of the other, character-generating keys, then the key handler derives a buffer code for the key, and inserts that into the keyboard buffer. The code entered into the buffer is derived from a table, which maps keycodes into ASCII codes, using the state of the various shifting and locking keys to alter the code if appropriate. In addition, the key-press is recorded in a 'last key pressed' location. This is to enable auto-repeating keys to be implemented, as described below.

For the standard keys, eg the letters, digits, punctuation marks etc, the buffer code is the ASCII code of the symbol. Thus when the code comes to be removed from the keyboard buffer (by `OS_ReadC`, for example), it is returned directly to the user. The other keys, such as the function keys and cursor keys, are entered as top-bit set characters, in the range `&80 - &FF`.

Interpreting buffer codes

When one of the top-bit-set codes is removed, it is not passed back to the caller. Instead, it is interpreted in a way determined by one of eight status bytes. Each byte controls the interpretation of a range of 16 codes: `&80 - &8F`, `&90 - &9F`, etc. Possible interpretations are:

- Ignore the code altogether
- Treat the code as a function key
- Return the key as two ASCII bytes
- Return the key as one ASCII byte.

Details of each of these interpretations is given in the appropriate `OS_Byte` descriptions.

A further complication is involved when the buffer code is that of a cursor key. These have codes `&XY`, where X is 8, 9, A or B (for normal, `Shifted`, `Ctrl'd` and `Ctrl Shift'd` cursor keys), and Y is B, C, D, E or F for `Copy`, and `←`, `→`, `↓`, `↑` respectively.

When one of these codes is read, the action depends on the cursor key status byte, set by `OS_Byte &04`. If this is zero, the code is not returned to you but causes the

input cursor to be moved in the appropriate direction, or a character to be read from the screen and returned to the caller.

If the cursor status byte is 1, then an ASCII code is returned, being $&80+(Y-4)$, where Y is as above. In effect, the arrow keys return the same codes as back space, horizontal tab, line feed and vertical tab, but with the top bit set. `[Copy]` returns a top-bit-set version of BELL (ASCII 7).

If the cursor status is 2, then the cursor key is interpreted as a function key, and is dealt with in one of the four ways noted above.

Before any of the above occurs, the input routine checks for a pending `[Alt]`. This happens when you press `[Ctrl][Alt][Shift]`. Such an occurrence is stored as a flag in the keyboard's status byte. If this flag is set when a byte is removed from the input buffer, the code is ORed with $&80$, and it is then passed back to the user with no further processing. This allows top-bit-set ASCII codes to be entered directly from the keyboard, by pressing (and releasing) the three keys, then typing the code which obtains the top-bit-set character when ORed with $&80$.

How function keys work

If a top-bit-set buffer code is read (it should be interpreted as a function key according to the appropriate status byte), then a special action is taken by the OS. First of all, it looks up the value of the OS variable which corresponds to the function key. The function key number is the lower nibble of the buffer code. So if the buffer code is $&81$, and codes in the range $&81$ to $&8F$ are treated as function keys, the variable read is `Key$1`.

The value is converted into a string (so if it was set using `*SETMACRO`, it is `OS_GSTransed`), and stored in a buffer. If the value was a null string (the function key wasn't set), the OS starts again, and removes the next character from the input buffer.

If the key variable was set, the first character is removed from the buffer where it was stored, and returned to the user. Characters read from this special function key buffer are simply returned; they are never interpreted in any way.

Subsequent calls to `OS_ReadC` and `OS_Byte` spot the fact that a function key is being read, and remove characters from the function key buffer instead of looking in the input buffer. This continues until the last character has been read from the key string. Input then reverts to the appropriate input buffer.

An `OS_Byte` (&D8 or 216) reads the readable number of characters from the currently active key string. If this is zero, then no function key is active. By setting this variable to zero, you can 'cancel' a function key, so that the next attempt to read a character accesses the input buffer.

Function key strings are programmed using the `*KEY` command. Alternatively, you can use the variables `Key$0` to `Key$15`, setting them using `*SET`, `*SETMACRO` or `*SETEVAL`. `*SETMACRO` has the advantage that the string returned by the function key can be made to vary from call to call.

Scanning the keyboard

You can read the status of keys, regardless of what is in the keyboard buffer, using a couple of `OS_Bytes`. These 'scan' the keyboard: they check whether you are pressing a particular key at the instant they are called, and return an indication of whether the key is down or not. Another related call scans the keyboard for any key presses and returns the code of the first key (in scanning order) to be pressed.

The keycodes used by these calls are neither ASCII nor the ones sent from the keyboard and subsequently handed to the keyboard handler. Rather, they are codes which are compatible with the internal key numbers on the BBC Micro and Master 128 series of micros. They take two forms. Internal keycodes are positive numbers, in the range 0 – 15 for 'special' keys, and 16 – 124 for the rest of the keys. 'Negative INKEY' numbers are those quoted when you call `OS_Byte` to look for particular keypress. There is a simple relationship between the two forms:

negative INKEY code = NOT internal key code
internal key code = NOT negative INKEY code

See `OS_Byte` &78 for a list of keys in internal key code order, and `OS_Byte` &81 (below) for a list in 'ASCII' order.

Auto-repeat and two-key rollover

As noted above, when a key is pressed, its code (the BBC-compatible internal key code plus 128) is stored in a location. This is called the 'last key pressed' location. If the key is still down two centi-second interrupts later (see below), its buffer code is inserted into the keyboard buffer. At the same time, the auto-repeat counter location is initialised to the auto-repeat delay value.

Every centi-second, an interrupt occurs on the Archimedes. During the servicing of this interrupt, the OS checks to see if the last key pressed is still held down. If it is, the auto-repeat counter is decremented. When it reaches zero, the code for the key is inserted into the buffer again. The auto-repeat counter is then reloaded from the current auto-repeat rate value. Thereafter, every time the counter reaches zero, the buffer code is inserted and the counter reloaded.

Every time a character is inserted into the keyboard buffer, during the centi-second interrupt, the inserted value may be altered according to the current state of the shift and locking keys.

There are, in fact, two 'last key pressed' locations. If a second key is pressed while the first is still down, the more recent one becomes the 'last key pressed', and when the original key is released, it does not cause the auto-repeat of the second key to stop. This is called two-key rollover.

The escape condition

The escape condition is an important state which alters the way in which several OS routines behave. This occurs when you press the current escape key, which is usually the one marked `Escape` on the keyboard; however, the key which causes an escape condition may be programmed. You can also cause an escape condition directly using an `OS_Byte`.

If enabled, an escape condition is brought about by pressing `Escape`, and results in an escape event. Setting the escape flag explicitly will not cause an escape event.

When an escape condition exists, the input routines `OS_ReadC` and `OS_Byte &81` do not wait for a character to appear in the input buffer. Instead, they return

immediately with some indication of the escape (C=1 and R2=&1B respectively). Similarly, if the screen is in paged mode and is waiting for Shift to be pressed, an escape condition will abandon the wait so that the screen scrolls freely. Finally, if the OS is waiting in a loop for space to appear in an output buffer (eg the printer), this wait is abandoned.

The idea of the escape condition, then, is for control to return as quickly as possible to the user, so that he or she can do something about it. There are two main ways of handling the escape. The condition can be simply cleared, so that the operation of OS_ReadC etc. reverts to normal. Alternatively, the condition can be 'acknowledged'. This clears the escape, but also performs various side-effect actions, such as flushing buffers, closing any *EXEC file etc.

You can test for an escape condition explicitly, using OS_ReadEscapeState. This is important if a program is executing in a loop which the user may want to escape from, but isn't performing any input operations which would let it know about the escape. It is also possible to disable escape conditions altogether.

Keyboard OS_Byte calls

OS_Byte &04 (04) – Cursor key status

On entry: R1 = action code (0 – 2)






On exit: R1 = old value
R2 is undefined

This call alters the status of the cursor editing keys, ie the four cursor control keys and Copy, according to the value in R1:

Value	Action
0	Enables cursor editing. This is the default state
1	Disables cursor editing. When pressed the keys return ASCII values
2	Cursor keys act as function keys

The ASCII values returned when R1=1 are:






Key **ASCII code**

	135
	136
	137
	138
	139

Note that the values for the arrow keys are the same as the corresponding ASCII control codes (line feed etc) with the top bit set.

If R1=2, the function key numbers assigned to the keys are:

Key **Function key number**

	11
	12
	13
	14
	15

The following OS_Byte also may be used to read or set this variable.

OS_Byte &ED (237) – Read/write cursor key status

On entry: R1 = 0 or new status
R2 = 255 or 0

On exit: R1 = previous status
R2 = value of next location (numeric keypad interpretation)

OS_Byte &0B (11) – Write keyboard auto-repeat delay

On entry: R1 = delay period

On exit: R1 = old delay period
R2 is undefined

You must hold down each key on the keyboard for a number of hundredths of a second (as set initially by *CONFIGURE Delay) before it begins to auto-repeat. This call enables you to change the initial delay from the default (until the next reset or power off). The delay is altered by R1 as follows:

Value	Effect
0	Disable auto-repeat
n	Set the auto-repeat delay to 'n' hundredths of a second

This variable may also be read and set using OS_Byte &C4 (196):

OS_Byte &C4 (196) – Read/write keyboard auto-repeat delay

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location (keyboard auto repeat rate)

OS_Byte &0C (12) – Write keyboard auto-repeat rate

On entry: R1 = repeat rate

On exit: R1 = old repeat rate
R2 is undefined

Unless auto-repeat has been disabled, each key on the keyboard will auto-repeat (after the auto-repeat delay time has elapsed, see above) at the rate specified by *CONFIGURE Repeat. This call enables the auto-repeat rate to be changed (until the next reset or power off). One particular use of this is to increase the rate to speed up cursor editing. The rate is altered by R1 as follows:

Value	Effect
0	Reset the auto-repeat and delay rate to their configured settings
n	Sets the auto-repeat rate to 'n' hundredths of a second

This variable may also be read and set using OS_Byte &C5 (197). See below.

OS_Byte &C5 (197) – Read/write keyboard auto-repeat rate


On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location (*EXEC file handle)

OS_Byte &12 (18) – Reset function keys

On entry: –




On exit: R1 is undefined
R2 is undefined

The strings assigned to function keys can be cleared individually by typing *KEY n . The present call clears all the function key definitions at once so that none of the function keys returns the strings previously assigned to it. This call also removes all of the Key\$n variables, so is equivalent to *UNSET Key\$*. Finally, it cancels any key string currently being read.

OS_Byte &76 (118) – Reflect keyboard status in LEDs

On entry: –

On exit: R2 is undefined

The settings of ,  and  are held in a location referred to as the keyboard status byte (see OS_Byte &CA (202)).

Under normal circumstances they are shown by the keyboard LEDs which are set into the keycaps. However, it is possible to write to the keyboard status byte directly without the LEDs changing accordingly. Calling OS_Byte &76 ensures that the current contents of the keyboard status byte are reflected in the LEDs.

OS_Byte &78 (120) – Write keys pressed information

On entry: R1 = internal key number of most recent key
R2 = internal key number of original key

On exit: –








This call simulates a key being pressed. It writes to the locations used to control auto-repeat and two-key rollover described above.

You can either set the last key pressed, simulating one key press, or the last key and original key pressed, ie simulating a second key press while the first is still held down. To simulate a single key press, the key's internal (BBC compatible) code plus 128 should be in R1 on entry, and R2 should contain 0. Alternatively, R1 should contain the second key press and R2 the original key code (both plus 128).

Note that if you set both the original and most recent keys, then subsequent actual key presses will be ignored. This is because the OS 'thinks' that there are two keys down already and so ignores subsequent actual presses until another OS_Byte &76 clears the key press information (with R1 and R2 both 0).

Below is a table of internal key numbers. You should not use codes below 16 in conjunction with the current OS_Byte. However, these codes do have a use in the 'scan keyboard' OS_Byte &79 (121).

Key	Internal key number
Shift (either or both)	0
Ctrl (either or both)	1
Alt (either or both)	2
Shift (left-hand)	3
Ctrl (left-hand)	4
Alt (left-hand)	5
Shift (right-hand)	6
Ctrl (right-hand)	7
Alt (right-hand)	8

Key	Internal key number
Left mouse button	9
Centre mouse button	10
Right mouse button	11
Q	16
3	17
4	18
5	19
	20
8	21
	22
-	23
	25
keypad 6	26
keypad 7	27
!	28
@	29
)	30
	31
	32
W	33
E	34
T	35
7	36
I	37
9	38
0	39
	41
keypad 8	42
keypad 9	43
	44 (but see OS_Byte &F7 (247) – it may cause a RESET)
Back tick/~	45
£/currency	46
Back space	47
1	48
2	49

Key	Internal key number
D	50
R	51
6	52
U	53
O	54
P	55
[56
↑	57
keypad +	58
keypad -	59
Enter	60
Insert	61
Home	62
Page Up	63
Caps Lock	64
A	65
X	66
F	67
Y	68
J	69
K	70
↵	73
keypad /	74
keypad .	76
Num Lock	77
Page Down	78
'/	79
S	81
C	82
G	83
H	84
N	85
L	86
;	87
]	88

Key	Internal key number
Delete	89
keypad #	90
keypad *	91
Tab	96
Z	97
Space Bar	98
V	99
B	100
M	101
,	102
.	103
/	104
Copy	105
keypad 0	106
keypad 1	107
keypad 3	108
Escape	112
f1	113
f2	114
f3	115
f5	116
f6	117
f8	118
f9	119
\	120
→	121
keypad 4	122
keypad 5	123
keypad 2	124

OS_Byte &79 (121) – Keyboard scan

On entry: R1 indicates key(s) to be detected

On exit: R1 indicates if/which key has been detected
R2 is undefined

This call enables you to check whether a particular key, or one of a range of keys, is currently depressed. The action depends on R1, as follows:

Value	Check for
-------	-----------

&00 – &7F	Lowest key number from R1 (for a range of keys)
-----------	---

&80 – &FF	Internal key number (R1 EOR &80) (for a single key)
-----------	---

The values returned are:

R1 > 0	Indicates that the key is depressed (for a single key)
--------	--

R1 = 255	Indicates no key is depressed (for a range of keys) or
----------	--

R1 =	(lowest) internal key number of the key(s) depressed
------	--

See the table above for the list of internal key numbers. Note that this call doesn't stop those key presses that are detected from being entered into the keyboard buffer.

OS_Byte &7A (122) – Keyboard scan from 16 decimal

On entry: –

On entry: R1 = internal key number of depressed key
R2 is undefined

This call performs a keyboard scan to see if any key is depressed (other than **Shift**, **Ctrl**, **Alt** and the mouse buttons) and if one is found, returns the internal key number of the lowest numbered one found or 255 if none is depressed. It is equivalent to OS_Byte &79 (121) with R1=16.

OS_Byte &7C (124) – Clear escape condition

On entry: –

On exit: –

This call clears any escape condition without further action.

OS_Byte &7D (125) – Set escape condition

On entry: –

On exit: –

This call simulates depression of the current escape key, setting the escape flag. However, an escape event is not generated.

OS_Byte &7E (126) – Acknowledge escape condition

On entry: –

On exit: R1 indicates if the escape condition has been cleared
R2 is undefined

This call attempts to clear an escape condition if one exists. It may or may not need to perform various actions to 'tidy up' after the escape condition depending on whether the escape condition side effects have been enabled or disabled (see OS_Byte &E6 (230)).

The contents returned in R1 indicate whether or not the escape condition has been cleared as follows:














Value	Meaning
255	The escape condition has been cleared
0	There was no escape condition to be cleared

OS_Byte &81 (129) – Scan a for a particular key

On entry: R1 NOT (internal key number)
R2 = &FF

On exit: R1 = &FF if key pressed, 0 otherwise
R2 = &FF if key pressed, 0 otherwise

To perform a keyboard scan for a particular character, the LSB of the negative INKEY value of the key is given in R1 (see table below) and R2 is set to 255. If both R1 and R2 contain &FF when the call returns, then the key being scanned is depressed.

Key	INKEY number
	-33
	-114
	-115
	-116
	-21
	-117
	-118
	-23
	-119
	-120
	-31
	-29
	-30
A	-66
B	-101
C	-83
D	-51
E	-35
F	-68
G	-84
H	-85
I	-38

Key	INKEY number
J	-70
K	-71
L	-87
M	-102
N	-86
O	-55
P	-56
Q	-17
R	-52
S	-82
T	-36
U	-54
V	-100
W	-34
X	-67
Y	-69
Z	-98
0	-40
1	-49
2	-50
3	-18
4	-19
5	-20
6	-53
7	-37
8	-22
9	-39
,	-103
-	-24
.	-104
/	-105
[-57
\	-121
]	-89
;	-88

Key	INKEY number
Escape	-113
Tab	-97
Caps Lock	-65
Scroll Lock	-32
Num Lock	-78
Break	-45
Back tick/~	-46
£/currency	-47
Back space	-48
Insert	-62
Home	-63
Page Up	-64
Page Down	-79
'/	-80
Shift (either or both)	-1
Ctrl (either or both)	-2
Alt (either or both)	-3
Shift (left-hand)	-4
Ctrl (left-hand)	-5
Alt (left-hand)	-6
Shift (right-hand)	-7
Ctrl (right-hand)	-8
Alt (right-hand)	-9
Space Bar	-99
Delete	-90
↵	-74
Copy	-106
↑	-58
→	-26
←	-122
↓	-42
keypad 0	-107
keypad 1	-108
keypad 2	-125
keypad 3	-109

Key	INKEY number
keypad 4	-123
keypad 5	-124
keypad 6	-27
keypad 7	-28
keypad 8	-43
keypad 9	-44
keypad +	-59
keypad -	-60
keypad .	-77
keypad /	-75
keypad #	-91
keypad *	-92
<input type="text" value="Enter"/>	-61
Left mouse button	-10
Centre mouse button	-11
Right mouse button	-12

As noted above, the negative INKEY number for a key can be derived from NOT (its internal key number), and vice-versa.

OS_Byte &B2 (178) – Read/write keyboard semaphore

On entry: R1 = 0 or new value of semaphore
R2 = 255 or 0

On exit: R1 = previous value of semaphore

This call allows you to enable or disable the processing of keyboard interrupts. If the flag is set to non-zero, then the OS will process any keyboard interrupts which occur. Setting it to zero causes the OS to ignore keyboard interrupts, effectively disabling the keyboard.

Similarly, by reading the status, you can ascertain whether the keyboard is currently enabled or not.

OS_Byte &C8 (200) – Read/write Break and Escape effect

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location (keyboard status)

This call provides a means of reading or resetting the effects of a RESET (including resets caused by **Break**) and of **Escape**.

The bit pattern in R1 determines the effects:

Setting	Effect
Bit 0 = 0	Normal escape action
Bit 0 = 1	Escape disabled unless caused by OS_Byte &7D (125)
Bit 1 = 0	Normal RESET action
Bit 1 = 1	Memory cleared on RESET (only if bits 2 – 7 are zero)

That is to say, a value of binary 0000001x causes a memory clear on reset, where x is the escape enabled/disabled bit.

OS_Byte &C9 (201) – Read/write keyboard disable flag

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location (keyboard status byte)

This call allows you to read and change the keyboard state (ie whether the keyboard is enabled or disabled). When it is enabled, all keys are read as normal. When it is disabled, the keyboard interrupt service routine does not place these keys in the keyboard buffer.

The main use of this call is by the NetFS *REMOTE command, which takes over the handling of key presses which come from some remote machine. The values for R1 are:

Value	Meaning
0	Enable keyboard input
Not 0	Disable keyboard input

OS_Byte &CA (202) – Read/write keyboard status byte

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location (RS423 input buffer space)

The keyboard status byte holds information on the current status of the keyboard, such as the setting of Caps Lock. This call enables you to read and change these settings.

The bit pattern in R1 determines the settings:

Bit	Indication when set
0	Pending <u>Alt</u>
1	<u>Scroll Lock</u> engaged
2	<u>Num Lock</u> disengaged
3	<u>Shift</u> depressed
4	<u>Caps Lock</u> disengaged
5	Always set
6	<u>Ctrl</u> depressed
7	<u>Shift Lock</u> enabled

If Caps Lock is engaged, then all characters are, usually, produced in upper-case whether Shift is pressed or not. If Shift Caps Lock is enabled, then pressing Shift and another key reverses the effect of the Caps Lock and produces a lower-case character.

See also OS_Byte &76.

OS_Byte &D8 (216) – Read/write length of function key string

On entry: R1 = 0 or new length
R2 = 255 or 0

On exit: R1 = previous length
R2 = value of next location (paged mode line count)

The location accessed by this call holds a count of the number of characters left in the currently active function key definition. If, on reading, it is non-zero, a function key is active. Conversely, it may be set to 0 to cancel any active function key. You should never write a non-zero value.

OS_Byte &DB (219) – Read/write Tab key code

On entry: R1 = 0 or new code
R2 = 255 or 0

On exit: R1 = previous code
R2 = value of next location (escape character)

By default, `[Tab]` generates ASCII 9. This call provides a means of reading the ASCII code currently assigned to `[Tab]`, and of changing it to another value.

If the code assigned to `[Tab]` is greater than or equal to &80, then `[Shift]` and `[Ctrl]` modify the value put into the keyboard buffer as follows:

`[Shift]` EORs the value with &10

`[Ctrl]` EORs the value with &20

Thus setting the code to &80+n makes the `[Tab]` key act exactly as function key 'n'.

If the assigned code is less than &80, then it cannot be modified by `[Shift]` or `[Ctrl]`.

OS_Byte &DC (220) – Read/write escape character

On entry: R1 = 0 or new code
R2 = 255 or 0

On exit: R1 = previous code
R2 = value of next location (interpretation of values 192 – 207)

Escape (ASCII 27) is the default escape character. This call enables you to read and redefine the buffer code of the current escape character. Whenever this code is inserted into the input buffer, an escape condition arises, if enabled by the various flags.

OS_Byte &DD (221) – Read/write interpretation of input values &C0 – &CF

OS_Byte &DE (222) – Read/write interpretation of input values &D0 – &DF

OS_Byte &DF (223) – Read/write interpretation of input values &E0 – &EF

OS_Byte &E0 (224) – Read/write interpretation of input values &F0 – &FF

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location

These calls enable you to read and change the way in which four groups of 16 input buffer codes in the ranges &C0 – &CF, &D0 – &DF, &E0 – &EF and &F0 – &FF respectively are interpreted.

The possible interpretations depend of the value of R1:

Value	Interpretation
0	Ignore the code
1	Code generates the string assigned to function key (code MOD 16)
2	Code generates a NULL (ASCII 0) followed by code
3 – &FF	Code generates the value R1 + (code MOD 16)

These buffer values are obtained either by inserting the code directly into the buffer (using OS_Byte &8A), or (in the case of RS423 input) by sending them to the RS423 port.

Additionally, the following function key presses produce certain codes in the range:

Key	Code	+ Shift	+ Ctrl	+ Shift Ctrl
f10	&CA	&DA	&EA	&FA
f11	&CB	&DB	&EB	&FB
f12	&CC	&DC	&EC	&FC
Insert	&CD	&DD	&ED	&FD

The default values for these variables are 1, &D0, &E0, and &F0 respectively. This means that the first set of codes act as function key presses, and the remainder have their input buffer code converted directly to the same ASCII value when read.

Note that when a reset occurs, the code &CA is inserted into the input buffer. This causes the key definition for function key 10 to be used for subsequent input (if key 10 is defined).

- *Note:* if any of these status bytes is set to 2 (to cause the return of a zero followed by another byte), then when **Ctrl**@ (ASCII 0) is pressed, it is read as two 0 bytes. This enables NULL to be detected without confusion with top-bit-set characters.

OS_Byte &E1 (225) – Read/write function key interpretation

OS_Byte &E2 (226) – Read/write Shift function key interpretation

OS_Byte &E3 (227) – Read/write Ctrl function key interpretation

OS_Byte &E4 (228) – Read/write Ctrl Shift function key interpretation

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location

These calls affect the interpretation of four sets of 16 buffer code values, in the ranges &80 – &8F, &90 – &9F, &A0 – &AF, &B0 – &BF. These codes are those produced by the function keys **F1** – **F9**, the cursor keys, and the **Print** key. Therefore, with these OS_Bytes you can control the action of those keys when pressed alone and in conjunction with **Shift** and **Ctrl**.

The possible interpretations depend of the value of R1:

Value	Interpretation
0	Ignore the key depression
1	Key generates the string assigned to function key (code MOD 16)
2	Key generates a NULL (ASCII 0) followed by code (see table)
3 – &FF	Key generates the value R1 + (code MOD 16)

The codes produced by the function keys are summarised in the table below:

Key	Code	+ [Shift]	+ [Ctrl]	+ [Shift][Ctrl]
[Print]	&80	&90	&A0	&B0
[f1]	&81	&91	&A1	&B1
[f2]	&82	&92	&A2	&B2
[f9]	&89	&99	&A9	&B9
[Copy]	&8B	&9B	&AB	&BB
[←]	&8C	&9C	&AC	&BC
[→]	&8D	&9D	&AD	&BD
[↓]	&8E	&9E	&AE	&BE
[↑]	&8F	&9F	&AF	&BF
[Page Down]	&9E	&8E	&BE	&AE
[Page Up]	&9F	&8F	&BF	&AF

The default values for these variables are 1, &80, &90, and 0. This means that function key presses return the appropriate key definition, **[Shift]** plus a function key returns an ASCII value between &80 and &8F, **[Ctrl]** plus a function key returns an ASCII value between &90 and &9F, and **[Ctrl][Shift]** plus a function key is ignored.

You should note the use of R1=2 for software which is capable of dealing with top-bit-set ASCII characters, ie the international character set in the range &A0 – &FF (160 – 255). It is recommended that you set function keys to return a NULL followed by the key code, so that they can be distinguished from actual ASCII characters which have been typed using, for example, **[Ctrl][Shift][Alt]** followed by another character. This also applies to the previous set of four OS_Byte calls.

- *Note:* if any of these status bytes is set to 2 (to cause a zero followed by another byte to be returned), then when **[Ctrl]@** (ASCII 0) is pressed, it is read as two 0 bytes. This is to enable NULL to be detected without confusion with top-bit-set characters.

OS_Byte &E5 (229) – Read/write Escape key status

On entry: R1 = 0 or new status
R2 = 255 or 0

On exit: R1 = previous status
R2 = value of next location (escape effects)

This call allows you to enable or disable the generation of escape conditions, and to read the current setting. Escape conditions may be caused by the depression of the current escape character or by the insertion of the corresponding buffer code in the input buffer, if placed there using OS_Byte &99 (153).

Values of R1 are as follows:

Value	Meaning
0	Enable escape condition generation
Not 0	Disable escape generation

When escape conditions are disabled, the current escape character generates its ASCII code instead.

The generation of escape conditions can also be disabled by OS_Byte &C8 (200).

OS_Byte &E6 (230) – Read/write escape effects

On entry: R1 = 0 or new status
R2 = 255 or 0

On exit: R1 = previous status
R2 is undefined

By default, the acknowledgement of an escape condition produces the following effects:

- Flushes all active buffers
- Closes any currently open *EXEC file
- Clears the VDU queue
- Clears the VDU line count used in paged mode
- Terminates the sound being produced.

This call enables you to determine whether the escape effects are currently enabled or disabled, and to change the setting if required.

The interpretation of R1 is as follows:

Value	Meaning
0	Enable side effects
Not 0	Disable side effects

OS_Byte &EE (238) – Read/write numeric keypad interpretation

On entry: R1 = 0 or new status
R2 = 255 or 0

On exit: R1 = previous status
R2 is undefined

This call controls the code which is inserted into the input buffer when you press one of the keypad keys. The inserted buffer code is derived from the sum of a base value (set by this call) and an offset, which depends on the key pressed. The inner (lighter) keys have two different offsets. The offset used depends on the state of the Num Lock key.

By default, the base number is 48, ie they generate codes which are displacements from 48 (ASCII '0').

The table below shows the offsets if `Num Lock` is on. In brackets are the characters which result when the base for the keypad is its default value of 48.

	-1 (/)	-6 (*)	-13 (#)
+7 (7)	+8 (8)	+9 (9)	-3 (-)
+4 (4)	+5 (5)	+6 (6)	-5 (+)
+1 (1)	+2 (2)	+3 (3)	
			-35 (Return)
+0 (0)		-2 (.)	

If `Num Lock` is off the offsets are:

	-1 (/)	-6 (*)	-13 (#)
-18 (home)	+95 (up)	+111 (page up)	-3 (-)
+92 (left)	(ignored)	+93 (right)	-5 (+)
+91 (copy)	+94 (down)	+110 (page down)	
			-35 (Return)
+157		+79 (delete)	

The keys labelled 'up' etc. refer to the cursor actions which result when `Num Lock` is off.

Unlike the function keys, you can set the numeric keypad base number to any value in the range 0 – 255. (If a generated code lies outside this range it is reduced MOD 256.)

Note that the code produced is modified if either **Shift** or **Ctrl** is pressed. See OS_Byte &FE (254) for details.

OS_Byte &F7 (247) – Read/write Break key actions

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 is undefined

This call controls the result of pressing the **Break** key ie how it affects the operation of the machine. The byte affected by the call consists of 4 two-bit numbers. Bits 0,1 control **Break**; bits 2,3 control **Shift Break**; bits 4,5 control **Ctrl Break**, and bits 6,7 control **Ctrl Shift Break**.

Each two-bit number may take on one of these values:

Value	Effect
0	Perform reset
1	Perform Escape
2	No effect
3	Undefined

The default value is &01, so **Break** causes an Escape condition, and all other combinations cause a reset.

OS_Byte &FD (253) – Read last break type

On entry: R1 = 0
R2 = 255

On exit: R1 = break type
R2 = value of next location (effect of **Shift** on keypad)

This call returns the type of the last break performed in R1:

Value	Break type
0	Soft break
1	Power-on reset
2	Hard break

OS_Byte &FE (254) – Set effect of Shift Ctrl on numeric keypad

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location (read/write startup options)

This call allows you to enable or disable the effect of **Shift** and **Ctrl** on the numeric keypad or to read which is the current state. These keys may modify the code just before it is inserted into the input buffer.

R1 is interpreted as follows:

Value	Meaning
0	Enable the effect of Shift and Ctrl
Not 0	Disable the effect of Shift and Ctrl

If the effect is enabled then the following actions occur:

- if the value \geq &80:
 - Shift** EORs the value with &10
 - Ctrl** EORs the value with &20
- if the value $<$ &80:
 - Shift** and **Ctrl** still have no effect

Keyboard SWI calls

OS_ReadEscapeState &2C (44)

On entry: -

On exit: C is set if escape is set, and clear otherwise

OS_ReadEscapeState sets or clears the carry flag depending on whether escape is set or not. Once an escape condition has been detected (either through this call or, for example, OS_ReadC returning with C=1), it should be acknowledged using OS_Byte &7E (126).

Note that OS_ReadEscapeState may be called from an interrupt routine. However, OS_Byte &7E may not be, so if an escape is detected under interrupts, the interrupt routine should set a flag which is checked by the foreground task, rather than attempt to acknowledge the escape itself.

OS_InstallKeyHandler &3E (62)

On entry: R0 = address of keyboard handler, or 0 for read-only

On exit: R0 = address of old keyboard handler

OS_InstallKeyHandler installs a new keyboard handler to replace the default code. It must be called with IRQs disabled, unless the read-only option is used.

The RS423 port

This section describes the operation of the RS423 input port. It also details the OS calls (all of them OS_Bytes) which control the general behaviour of the RS423 port in both directions.

As already noted, you can take input from the RS423 input buffer, and ignore the keyboard buffer, or read characters from the keyboard buffer and ignore the RS423. In the latter case, you have the choice of disabling the RS423 port entirely, so no characters are received, or having incoming characters buffered under interrupts.

When the RS423 port is used as the input stream, the calls `OS_ReadC` and `OS_Byte` are affected: they cause characters to be removed from the RS423 input buffer instead of the keyboard buffer. However, the action of the scanning-type input functions, eg `OS_Byte` with a negative `INKEY` argument, are not affected; they always use the keyboard.

Interpretation of RS423 characters

You can decide whether characters read from the RS423 buffer will be treated exactly as those read from the keyboard buffer, or whether all RS423 characters will be treated as pure ASCII.

The difference is important when top-bit-set characters are read. As discussed in the previous sections, characters between `&80` and `&FF` read from the keyboard buffer may be handled in a variety of ways. This depends on the exact range that the character is in, and on the state of the eight interpretation flags. For cursor key codes, it also depends on the cursor key interpretation flag.

Usually, top-bit-set characters read from the RS423 buffer are not treated specially. For example, if the remote device sends the code `&85`, when this is read (using `OS_Byte` or `OS_ReadC`), that ASCII code will be returned to the caller immediately. It is sometimes useful to be able to treat RS423 characters in exactly the same way as keyboard characters. `OS_Byte` (181) allows this.

Other RS423 functions

`OS_Bytes` are also provided to perform the following RS423 functions:

- Set and read the RS423 data rates
- Disable the RS423 input processing
- Read and set the RS423 control register
- Read and set the RS423 handshake extent value.

RS423 OS_Byte calls

These calls are documented below.

OS_Byte &07 (7) – Write RS423 receive rate

On entry: R1 = baud rate code

On exit: R1 is undefined
R2 is undefined

This call sets the RS423 baud rate for receiving data according to R1, as follows:

Value	Rate
0	9600 baud
1	75 baud
2	150 baud
3	300 baud
4	1200 baud
5	2400 baud
6	4800 baud
7	9600 baud
8	19200 baud

The default speed is the speed set by *CONFIGURE Baud. OS_Byte &08 sets the RS423 transmit rate. OS_Byte &F2 (242) may be used to read both the receive and transmit data rates.

- *Note:* if the receive rate is set differently from the transmit rate, then reception at 9600 and 19200 baud rates is not guaranteed. This is because these rates are generated by a different timer from the transmit ones, and have errors of -7% and +8.5% respectively. The maximum permitted error for reliable RS423 operation is +/-5%.

OS_Byte &9C (156) – Read / write asynchronous communications state

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = old value

This call accesses the control byte of the RS423 port. It acts in a similar fashion to the OS_Bytes in the range &A6 – &FF, ie performs the operation:

new value = (old value) AND R2 EOR R1

and returns the old value in R1. However, in addition to updating the status byte in RAM, it also updates the hardware register which controls the RS423 characteristics.

The call enables the current settings of the transmitter, receiver, interrupts and the RS423 Request To Send (RTS) to be read or altered.

When writing, the effect depends on the bits in R1:

Bit 1	Bit 0	Effect
0	0	No effect
0	1	No effect
1	0	No effect
1	1	Reset transmit, receive and control registers

Bit 4	Bit 3	Bit 2	Word length	Parity	Stop bits
0	0	0	7	even	2
0	0	1	7	odd	2
0	1	0	7	even	1
0	1	1	7	odd	1
1	0	0	8	none	2
1	0	1	8	none	1
1	1	0	8	even	1
1	1	1	8	odd	1

Bit 6	Bit 5	Transmission control
0	0	RTS low, transmit interrupt disabled
0	1	RTS low, transmit interrupt enabled
1	0	RTS high, transmit interrupt disabled
1	1	RTS high, transmit interrupt disabled

Bit 7 Receive interrupt

0	Disabled
1	Enabled

The default setting for bits 2 – 4 comes from the *CONFIGURE Data value, shifted left by two bits. The current value of this byte may be read (but not set) using OS_Byte &C0 (192).

OS_Byte &C0 (192) – Read RS423 control byte

On entry: R1 = 0
R2 = 255

On exit: R1 = previous value
R2 = value of next location (flash counter)

OS_Byte &B5 (181) – Read/write RS423 input interpretation status

On entry: R1 = 0 or action code (0 or 1)
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location (NoIgnore state)

RS423 input can be altered between two states, either its default state in which:

- the current escape character is ignored
- the function key codes are not expanded
- events are not generated;

or the alternative state, in which RS423 input:

- behaves just like the keyboard

The value in R1 has the following interpretations:

Value	Interpretation
0	Simulate keyboard input
1	Set default status

OS_Byte &BF (191) – Read/write RS423 busy flag

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 contains the previous state
R2 is undefined

This call provides a means of reading or resetting the RS423 busy flag which is used to ensure that the serial interface hardware is not taken over whilst it is currently in use.

It is provided mainly for historical reasons: the cassette interface and RS423 port shared the same hardware on the BBC/Master 128 machines.

Bit 7 of R1 has the following interpretations when read and written:

Value	Meaning
Clear	Indicates RS423 is busy ($0 \leq R1 < \&80$)
Set	Indicates RS423 is free ($\&80 \leq R1 < \&100$)

OS_Byte &CB (203) – Read/write RS423 input buffer minimum space

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 = value of next location (RS423 ignore flag)

By default, the RS423 input routine attempts to halt input (by de-asserting the RTS line) when there are 9 free spaces left in the input buffer. This call allows the value at which input is halted to be read or changed if necessary.

OS_Byte &CC (204) – Read/write RS423 ignore flag

On entry: R1 = 0 or new value
R2 = 255 or 0

On exit: R1 = previous value
R2 is undefined

This call is used to read or change the flag which indicates whether RS423 input is to be ignored. Although this call can stop data being placed in the RS423 input buffer, data is still received normally and errors will still generate events (if enabled). The flag's meaning is as follows:

Value	Meaning
0	Enable RS423 input
Not 0	Disable RS423 input

OS_Byte &F2 (242) – Read RS423 baud rates

On entry: R1 = 0
R2 = 255

On exit: R1 = encoded RS423 baud rates
R2 = value of next location (timer switch state)

On exit R1 contains an encoded value which gives the baud rate for RS423 receive and transmit. Bits 0 – 2 of the result give the transmit baud rate, and bits 3 – 5 give the receive rate. Bit six is always set. The two 3-bit values are encoded as follows:

Value	Rate
0	19200
1	1200
2	4800
3	150
4	9600
5	300
6	2400
7	75

Thus a value of &4D means that the transmit rate is 300 baud, and the receive rate is 1200 baud.

THE *EXEC INPUT STREAM

The *EXEC file is the ‘half a stream’ mentioned at the start of this chapter. It is not a proper stream because you cannot use OS_Byte &02 to select between it and the other two input streams. Once a *EXEC file has been activated, it will override any other input until it is exhausted or is closed explicitly.

To open a *EXEC file, you use the command

```
*EXEC filename
```

where filename must exist. This file is opened for input. From then, any request for characters by OS_ReadC or OS_Byte &81 will read from the file instead of the keyboard or RS423. No interpretation is placed on the character at all; it is passed directly back to the caller.

After the last character has been read from a *EXEC file, the file is closed automatically and input reverts to the previous input stream. Alternatively, you can

close the file by issuing another *EXEC command (with no filename), or by acknowledging any pending escape condition.

The OS determines if there is an active *EXEC file by reading a memory location. This contains zero, if there is no *EXEC file open, or the handle of the file if there is. You can access this location using the OS_Byte documented below. This enables you to:

- determine for yourself if there is an open *EXEC file
- control the *EXEC file independently of the command line interface.

***EXEC file OS_Byte call**

OS_Byte &C6 (198) – Read/write *EXEC file handle

On entry: R1 = 0 or new handle
R2 = 255 or 0

On exit: R1 = previous handle
R2 = value of next location (*SPOOL file handle)

It can be used to disable a *EXEC file temporarily (by setting the handle to 0) without closing it. If you are changing the file handle, the new file must be open for input or update, otherwise a Channel error occurs. If an attempt is made to use a write-only file for the *EXEC file, a Not open for reading error is given.

THE COMMAND LINE INTERPRETER

As explained in the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS** there are two ways in which you can interact with the OS and the various modules which provide extensions to it. The first way is to call one of the many SWI routines provided, such as `OS_Byte`, `OS_ReadMonotonicTime`, `Wimp_Init` etc. The SWI interface provides an efficient calling mechanism for use within programs in any language.

However, for users wishing to issue commands to the operating system, the SWI interface is not so convenient. As it is difficult to remember SWI names, reason codes, register contents on entry and exit, etc, the command line interpreter (CLI) interface is often used. Using this technique, you enter a textual command string, possibly followed by parameters, which is then passed by the application to the OS. The OS tries to decode the command and carry out the appropriate action. If the command is not recognised by the OS, the other modules in the system try to execute the command instead.

The CLI interface is a powerful one because the OS performs a certain amount of pre-processing on the line before it attempts to interpret it. For example, variable names may be substituted in the parameter part of the line, and command aliases may be used.

By convention, an application passes commands to the OS if they are prefixed by the `*` character. For example, from the BASIC `>` prompt, any OS command may be issued simply by making `*` the first non-space character on the line. The `*` is not part of the command; the OS, in fact, strips any leading `*`s and spaces from a command before it tries to decode it.

Some languages also provide built-in statements which can be used to perform an OS command. Again, BASIC provides the `OSCLI` statement, which evaluates a string expression and passes this to the OS command line interpreter. The 'C' language provides the `system()` function for the same purpose.

CALLING THE COMMAND LINE INTERPRETER

As with all other interaction with the OS, you call the command line interpreter using one of the following SWI instructions:

OS_CLI &05 – Command Line Interpreter

This routine processes a command which is sent to it as a string terminated by a NULL (ASCII 0) or linefeed (ASCII 10) or carriage return (ASCII 13).

On entry: R0 = pointer to string

On exit: Action performed according to the command

THE ACTION OF OS_CLI

This section describes in detail what OS_CLI actually does to execute the command passed to it.

Check stack space

The OS needs a certain amount of workspace to deal correctly with a command. If this is not available, the error `No room on supervisor stack` will be generated.

Skips *s and spaces

Stars are skipped because you may sometimes type in a * before a command even when it isn't required. For example, all lines typed in response to the Arthur Supervisor prompt are passed straight to OS_CLI, so there is no need for you to prefix the command by *.

Check for comments

A comment is introduced by the symbol '!'. If one is present the OS ignores the string and returns immediately. Comment lines are useful in files of * commands which are *EXECed. Anything may follow the '!' (which must be the first non-space character on the line).

Check command length

A * command line must be less than or equal to 256 bytes long, including the terminating character. If it is not, the line is ignored. No error is generated.

Check for redirection

Redirection allows the input or output streams (or both) to be replaced for the duration of the command. They are routed to specified filenames instead of the usual devices. Output redirection can be viewed as having a *SPOOL file open for the duration of the command, and disabling all streams except for that one. Input redirection is like having a *EXEC file open for the duration of the command.

The format of a redirection is:

```
{redirection spec}
```

where the redirection spec is at least one of:

```
> filename Output goes to filename  
< filename Input read from filename  
>> filename Output appended to filename
```

This redirection of input and output is terminated at the end of the OS_CLI call or on any error. Redirection can be specified in any part of the command, and after being decoded, it is stripped before the rest of the command is interpreted. For example:

```
*CAT { > mycat }  
*LEX { > printer: }  
*BASIC -quit { < answers } prog
```

Note that spaces in the redirection are important. There must be at least one space between all elements, otherwise it will not be recognised as redirection.

Check for single-character prefixes

These are as follows:

/	Equivalent to RUN
%	Skip alias checking
-	Filing system name, eg -adfs-
@	Perform command on host processor (if TUBE fitted)
.	Check for Alias\$. and use CAT if it doesn't exist

The use of % at the start of the line enables you to access a built-in command which has an alias currently overriding it. See below for an explanation of aliases.

Giving a filing system name at the start of a command causes that filing system to be active for the duration of the command. When the command terminates, the current filing system is re-selected. This facility is additional to the ability to specify filing system names in pathnames (see the chapter **FILING SYSTEMS**).

Check for aliases

An alias is a variable of the form Alias\$cmd, where cmd is the command name to match. If an alias exists which matches the current * command, the following takes place: the OS obtains the value of the variable and replaces any of %0 to %9 in the value by the parameters, separated by spaces, that it reads on the rest of the input line. %*n in an alias stands for the rest of the command line, from parameter 'n' onwards.

Any unused parameters, which are given, are directly appended to the alias. The OS then recursively calls OS_CLI for all lines in the expanded value. However, it may give up at this stage if either the stack or its buffer space becomes full. For example, suppose the command

```
*SETPS 0.235
```

is issued. Suppose further that a variable exists called Alias\$SETPS, and that this has the value -NET-PS %0|MCONFIGURE PS %0. The OS will match the command

name against the alias variable. It will then substitute all occurrences of %0 in the variable's value by 0.235. Then, the two lines of the variable will be executed thus:

```
-NET-PS 0.235  
CONFIGURE PS 0.235
```

So, the net effect of executing the original command is to set the network printer server both temporarily, and also in the permanent configuration.

Another example using the parameter substitution is

```
*SET ALIAS$MODE ECHO |<22> |<%0>
```

The '|'s before the angle brackets are to stop them from being evaluated when the *SET command is entered. Typing *MODE n will then set the display to mode 'n'.

Look-up the command in the OS table

The OS recognises several commands which it acts upon to perform a variety of tasks. These tasks fall into three main categories:

- Executing machine code routines
- Interrogating the settings of various machine parameters. For example, the current date and time as stored in the CMOS clock
- Resetting the values of certain machine parameters. For example, the keyboard auto-repeat rate.

Any command offered to the CLI, which the operating system doesn't recognise, is passed around to see if it is recognised by any other module. It will be passed round all the modules which are loaded into the machine, including the Filing System Manager which recognises general filing system commands such as *DIR.

Finally the Filing System Manager offers the command to the filing system active for this command, which will recognise, for example, *BACK. If the command is still unclaimed at this stage an attempt is made to *RUN a file of the name given.

A list of commands which are recognised by the operating system is given below. These commands are listed under the heading 'Arthur' 'MOS Utilities' in response to a *HELP COMMANDS command.

There are more commands recognised by the OS, which are not documented here. These are filing system-related commands, such as *SPOOL, and commands to do with relocatable modules. Descriptions of these commands may be found in the appropriate chapters.

Finally, there are many commands which aren't recognised by the OS at all, but are implemented by the various modules. See the appropriate chapters for descriptions of these commands.

Command	Description
*CONFIGURE	Define non-volatile RAM configuration settings
*ECHO	Reflect translated string to the screen
*ERROR	Generate an error with given number and text
*EVAL	Evaluate an expression and print the result
*FX	Alter OS 'parameters' by calling OS_Byte
*GO	Execute a machine code program
*GOS	Enter the * prompt supervisor
*HELP	Display help information
*IF	Execute commands conditionally
*IGNORE	Define printer ignore character
*KEY	Define soft key string
*SET	Assign a string to a system variable
*SETEVAL	Assign a value to a system variable
*SETMACRO	Assign an expression to a system variable
*SHADOW	Change screen bank on next mode change
*SHOW	Display variables
*STATUS	Display CMOS RAM configuration settings
*TIME	Display date and time from CMOS clock
*TV	Specify the vertical alignment and interlace option
*UNSET	Delete a variable

OPERATING SYSTEM COMMANDS

***CONFIGURE**

Syntax: *CONFIGURE [<param 1> [<param 2>]]

*CONFIGURE defines the configuration settings held in the battery-backed RAM. These are made current on initial power-on and after a hard break (**Ctrl** RESET), and do NOT take effect immediately.

If the command is given with no parameters at all, then the configuration options are listed.

If it is given with parameters then it is used to alter a particular setting. <param 1> identifies the setting to be changed, as defined below. <param 2> defines the value to be stored in the appropriate location in the battery backed RAM. Some settings have more than one <param 2>, and sometimes there are none at all.

Where a number is required, it may be given in decimal, as a hex number preceded by &#, or a number of the form base_num, where base is the base of the number in decimal in the range 2 to 36. For example 2_1010 is another way of saying 10.

The parameters are described below:

Baud <n>

Sets the default RS423 transmit and receive rate according the specified value <n> in the range 0 – 8:

Value	Baud rate
0	9600
1	75
2	150
3	300
4	1200
5	2400
6	4800
7	9600
8	19200

Boot

Reverses the actions of RESET and **Shift** RESET. See also NoBoot.

Caps

Sets the keyboard Caps Lock (ie caps lock is on) on reset. See also NoCaps and ShCaps.

Country <country>

Specifies the default country to be set for the International module. The countries are:

Master
Compact
UK
Italy
Spain
France
Germany
Portugal
Esperanto
Greece

See the documentation on **The international module** for more details.

Data <n>

Specifies the default data format used by the RS423 interface according to the specified value, in the range 0 – 7:

Value	Word length	Parity bits	Stop
0	7	even	2
1	7	odd	2
2	7	even	1
3	7	odd	1
4	8	none	2
5	8	none	1
6	8	even	1
7	8	odd	1

Delay <n>

Sets the keyboard auto-repeat delay to the specified number of hundredths of a second, from 0 to 255. This is the length of time for which a key has to be held down before auto-repeat starts. A value of 0 disables auto-repeat. The delay can be temporarily changed using *FX11. See also *CONFIGURE Repeat.

Dir

Causes the ADFS to initialise with the root directory, \$, selected. See also NoDir.

Drive <n>

Causes the machine to initialise with drive <n> selected. On a machine with both Winchester and floppy drives, <n> has the following meanings:

- n = 0 – 3 Select floppy drive 0 – 3
- n = 4 – 7 Select Winchester drive 0 – 3

DumpFormat <n>

Selects the format to be used by *DUMP, *LIST, *LIST commands and the VDU: output device. <n> is a four-bit number. The bottom two bits define how control characters are displayed, as follows:

Value	Meaning
0	OS_GSTrans format is used (eg A for ASCII 1)
1	Period (.) is used
2	<n> is used, where 'n' is in decimal
3	<&n> is used, where 'n' is in hex

If bit 2 is cleared, top-bit set characters are treated as control codes, otherwise they are treated as printable characters.

If bit 3 is set, characters are ANDed with &7F before being processed, otherwise they are left as they are.

File <n>

Defines the default filing system, where <n> is the filing system number. The current numbers are:

Number	Filing system
5	NetFS (Econet)
8	ADFS

This options is replaced in OS versions 0.40 and greater by the new setting FileSystem:

FileSystem <n> | <name>

If a number is given as the parameter, then the meaning is the same as that for the File option. Alternatively, the name of the filing system may be specified instead, eg ADFS, NET.

Floppies <n>

Causes the machine to initialise believing that <n> floppy drives are attached.

FontSize <n>

Reserves <n> * 4K pages for the font cache. If <n> = 0 then no space is reserved for fonts. The maximum space allowed is 255*4K.

FS [<nnn>.<sss> or <name>

Selects the number of the network fileserver. <sss> is the station number. <nnn> is the network number, which is optional. Alternatively, a file server name may be given.

HardDiscs <n>

Causes the machine to initialise believing that <n> Winchester discs are attached.

Ignore [<0 - 255>]

Defines the 'printer ignore character' (ie the character which is not to be directed to the printer), by means of its ASCII code. If the character code is omitted, all characters are sent to the printer when it is enabled. The character can be changed temporarily by the *IGNORE command.

Language <n>

This option is only available on OS versions greater than 0.30. The parameter is the module number of the language which should be started on a hard reset. Modules numbers are listed along with other information by the *MODULE command.

Lib 0 | 1

This option is only available on OS versions greater than 0.30. If the parameter is given as 0, then the default fileservr library will be used when you log on to a fileservr (usually \$.Library). If 1 is given as the argument, the library \$.ArthurLib will be used instead.

Loud

This sets full volume for the bell (**Ctrl**G) sound.

Mode <n>

Defines the default screen mode setting to be <n>.

MonitorType <n>

The parameter is the type of monitor fitted to the machine, as follows:

<n>	Monitor
0	50Hz 'TV' standard monitor
1	Multi-sync monitor
2	Hi-resolution 64 KHz monochrome monitor (400-series only)

Note that certain modes are only available with certain monitor types. See the chapter **THE VDU DRIVERS** for details.

NoBoot

Assigns the normal functions to RESET and **Shift**RESET.

NoCaps

Resets the keyboard **Caps Lock** (ie **Caps Lock** is off on reset).

NoDir

Causes ADFS to initialise with no directory selected. This prevents the disc from being accessed on initialisation.

NoScroll

Enables the scroll protect option. This relates to the VDU 23,16 option which prevents an auto-newline when a character is printed at the end of a line.

Print <n>

Defines the printer driver according to the value specified:

Value	Printer type
0	Sink (ie no output printed)
1	Printer port (ie parallel, Centronics type)
2	RS423 port (ie serial)
3	User printer driver
4	Network printer – handled through the Econet vector

PS [<nnn>.]<sss> or <name>

Selects the number of the network printer server. <sss> is the station number. <nnn> is the network number, which is optional. Alternatively you can give the name of a printer server.

Quiet

Sets half volume for the bell (**Ctrl**G) sound.

RamFsSize <n>

Sets the number of pages for the RAM filing system (when that module is present). The page size is 8K on 300 series machines and 32K on 400-series machines.

Repeat <n>

Sets the keyboard auto-repeat rate to the specified number of hundredths of a second. The rate can be changed temporarily using *FX12.

RMASize <n>

Reserves <n> pages (8K or 32K for 300 series and 400 series machines) in the relocatable module area (RMA) for modules. For OS versions prior to 0.3, <n> is the number of pages in total, with the system always reserving enough for the system modules to initialise.

For OS 0.3 and above, <n> is the number of extra pages to be allocated once the system modules have claimed their space.

The maximum value is 255, though 128 is the current limit imposed by the hardware.

ScreenSize <n>

Reserves <n> pages for the screen. The page size and default setting (when <n> = 0) depend on the series of the machine and its RAM size:

Model	RAM	Page	Default
305	0.5M	8K	80K (10 pages)
310	1.0M	8K	160K (20 pages)
410	1.0M	32K	160K (5 pages)
440	4.0M	32K	160K (5 pages)

Scroll

Disables the scroll protect option. See NoScroll.

ShCaps

Sets the keyboard **Shift** **Caps Lock** option so that alphabetic keys produce upper-case characters and **Shift**ed alphabetic keys produce lower-case characters. See NoCaps and Caps.

SoundDefault <spkr> <vol> <voice>

Sets the default sound characteristics. The <spkr> parameter is 0 to disable the internal speaker on reset, 1 to enable it. The <vol> parameter sets the overall sound volume, and is in the range 0 (min) to 7 (max). The <voice> parameter sets the voice which will be assigned to channel 1, and is in the range 1 – 16.

SpriteSize <n>

Reserves <n> (8K or 32K) pages for the sprite system. If <n> = 0 then no sprite space is reserved. The maximum allowed is 255 pages.

Step <n> [<drive>]

Sets the floppy disc drive step rate to <n>. If <drive> is omitted, the step for all floppy drives is set, otherwise just the one specified is changed. <n> gives the step time in milliseconds. The nearest value greater than or equal to the one requested will be used. The current values are 2, 3, 6 and 12ms.

Sync <n>

Selects either vertical only (<n> = 0) or composite (<n> = 1) sync on the 'VS' output of the video connector. The CMOS-RAM default is 1, and shouldn't be changed when using the supplied monitors.

SystemSize <n>

Reserves <n> pages for the system heap. The page size for a given RAM configuration is as for ScreenSize. If <n> = 0 then the default amount (32K) is reserved. The maximum allowed is 255.

TV <n> [[,]<m>]

Selects the default setting for the *TV command parameters. See that command's description for details of <n> and <m>.

*ECHO

Syntax: *ECHO <string>

*ECHO takes the string following it, translates it using OS_GSTrans and then displays it on the screen.

Example: *ECHO |G

This performs a **Ctrl**G and so emits the bell sound.

Example: *ECHO <Run\$Path>

This types the filing system path used for searching for commands.

See *SET below for more examples of how *ECHO may be used.

*ERROR

Syntax: *ERROR <num> <string>

*ERROR generates an error with the specified error number and text. The error message starts at the first non-space character after the number.

Example: *ERROR 100 No such file

This generates an error with error number 100 and the text No such file.

***EVAL**

Syntax: *EVAL <expression>

*EVAL evaluates the string following the command and prints out the result. The string is evaluated by OS_EvaluateExpression (&2D). See the description of that call in the chapter NUMBER CONVERSIONS for the full syntax of the expression.

Example: *EVAL (<fred>+532)/10

***FX**

Syntax: *FX <param 1> [[,]<param 2> [[,]<param 3>]]

Each *FX call executes the corresponding OS_Byte routine using the parameters specified to initialise the R0, R1 and R2 registers respectively. These parameters may be separated by commas or spaces. Unspecified parameters are taken to be zero.

***GO**

Syntax: *GO [<param 1>] [<args>]

*GO executes a machine code program.

If <param 1> is specified, *GO takes it to be the 32-bit address of the program's entry point, and starts execution of the application which starts at that point, with the processor in user mode and with interrupts enabled. The address may be given in any form which is acceptable to OS_ReadUnsigned. Note that the code called by this command is not expected to return, except perhaps via OS_Exit.

If <param 1> is not specified the address is taken to be &8000. In either case one or more arguments can be passed to the program.

*GOS

Syntax: *GOS

*GOS calls the operating system supervisor, from which * commands may be issued. You can return to the previous environment using the *QUIT command.

The advantage of issuing * commands from this prompt, instead of from an application's command mode (eg from the BASIC > prompt) is that when no application is active the memory map of the machine may be altered. This means that, for example, the RMA may be increased in size in response to a *RMLOAD command if it is currently too small. This can't be done when an application is active, and an error such as No room in RMA would be given.

*HELP

Syntax: *HELP <string>

*HELP can provide help information on the operating system commands, filing system commands and any commands declared by currently resident modules. The parameter can be a single command name, in which case details of the one command are given. A list of command names can also be given and information about each parameter in the list is displayed. In addition, the command name can be abbreviated with a '.', in which case all commands which match it are listed.

The information is displayed in paged mode, so you may have to press **Shift** to see all of the information.

Example: *HELP ECHO

This gives the syntax required by *ECHO and a brief description of its function.

Example: *HELP SLOAD SSAVE

This gives help information about the two sprite commands *SLOAD and *SSAVE.

Example: *HELP RM.

This gives information about all the commands beginning with the letters RM, ie the relocatable module commands.

Example: *HELP .

This gives information about all the commands which the system knows about.

*IF

Syntax: *IF <expr> THEN <command1> [ELSE <command2>]

*IF executes the first command <command1> if the result of the expression <expr> is true (ie non-zero). If the expression is false, it executes the command <command2> if the ELSE is present, otherwise, it does nothing. The expression is read using SWI OS_EvaluateExpression, and may use any operators and operands recognised by that routine.

*IGNORE

Syntax: *IGNORE [<param 1>]

*IGNORE sets the 'printer ignore character'.

It is sometimes necessary to prevent certain characters from being sent to the printer. This facility is most commonly used to 'filter out' line feed characters (ASCII 10) sent to printers which perform an automatic line feed when they are sent a carriage return. <param 1> is the ASCII code of the character to be ignored. If it is omitted, all characters are sent to the printer.

*KEY

Syntax: *KEY <key number> [<text>]

This command allows a string of text (of up to 255 characters in length) to be assigned to one of the 16 available function keys. Then, when the function key is pressed, its subsequent input from the keyboard is read from that string. This enables

a commonly used sequence of instructions to be assigned to a key so that just a single key press is required to issue them each time.

In addition to **f1** to **f2**, the function keys are **Print** (function key 0), **Copy** (key 11), and **←**, **→**, **↓** and **↑** (keys 12 to 15 respectively). To use the cursor keys as function keys, you must first issue the command `*FX4,2` or equivalent.

The string `<text>` is transformed by `OS_GSTrans` before being stored, and so may incorporate control characters, etc. If `<text>` is a null string, the definition for that key is reset so that pressing it has no effect on subsequent character input.

Function key definitions are normally unaffected by a soft break but are lost following a hard break.

Using `*KEY n <text>` is equivalent to `*SET KEY$n <text>`. That is to say, for each function key there is a corresponding variable. This enables a key's definition to be read before it is used, and generally manipulated in the way of any other variable. Also, because a key string can be set as a macro, its value may be made to change each time it is used.

Example: `*KEY 1 CHAIN "WELCOME"|M`

This makes pressing **f1** equivalent to typing:

```
CHAIN "WELCOME" ←
```

`*SET`

Syntax: `*SET <varname> <text>`

`*SET` can be used to assign a string, given by the parameter `<text>`, to a variable `<varname>`.

Example: `*SET HELLO "Hello and how are you today?"`

This will create the variable HELLO whose current value is the string Hello and how are you today? Many commands allow you to access a variable by placing its name within angled brackets, eg:

```
*ECHO <HELLO>
```

There are certain 'special' variables which are always present. You may alter the string assigned to them but you cannot delete them. These are:

Sys\$Time	Time in the form 12:03:12
Sys\$Date	Date in the form Mon, 12 July
Sys\$RCLimit	Maximum permitted return code
Sys\$ReturnCode	Most recent return code
Sys\$Year	Year in the form 1987

The return code variable is set either by a *SET command, or when an application uses the routine OS_Exit (see the chapter **THE PROGRAM ENVIRONMENT**).

If an attempt is made to set the variable Sys\$ReturnCode outside of the range 0 – Sys\$RCLimit, the error Return code limit exceeded is given. The 'error free' return code is 0.

The operating system extension modules, etc may all set up their own variables in this way. For example, the OS command line interpreter recognises the variable Cli\$Prompt. If this variable has been defined, it will use its value as the operating system prompt. If it is not set up, the default prompt will be used.

If the variable defined is given a name of the form Alias\$<name>, and the string assigned to it is the name of a command, then 'name' becomes an alternative name for the command. See the section above on the way in which the OS deals with commands for more details.

- *Note:* it is possible to create variables whose names are sequences of digits, eg *SET 12 HELLO. This is not recommended as you cannot then access the name using the <name> convention. <digits> means the character whose ASCII code is 'digits'.

***SETEVAL**

Syntax: *SETEVAL <varname> <expression>

*SETEVAL assigns the value of the variable <varname> to be the value of the expression <expression>. Variables created with this command will be of the type number, rather than string. See OS_EvaluateExpression for details of the syntax of <expression>.

Example: *SET rate 12
*SETEVAL rate rate+1
*SHOW rate

***SETMACRO**

Syntax: *SETMACRO <varname> <value>

*SETMACRO assigns an expression to a variable. The parameters making up the expression are not evaluated when the command is given. Instead, they are re-evaluated each time the variable is used.

Example: *SETMACRO CLI\$PROMPT <SYS\$TIME>

This resets the operating system prompt to be the current time whenever the prompt is given. This differs from *SET CLI\$PROMPT <SYS\$TIME> which sets the prompt to be the time when the *SET command was given, not when the prompt is printed.

***SHADOW**

Syntax: *SHADOW <value>

*SHADOW is provided for compatibility with the Master-128. If the value is omitted, or has the value 2, then bank 2 of screen memory is used by subsequent mode changes. If the value is 1 then bank 1 is used.

***SHOW**

Syntax: *SHOW [<[wildcarded] varname>]

*SHOW gives the name, type and current value of any variables whose name matches the string given. These include the 'special' variables listed in *SET above or any other variables which have been defined.

Example: *SHOW CLI\$PROMPT

This produces the following type of response:

```
CLI$Prompt : type String, value : Hi There
```

Example: *SHOW ALIAS\$AID

After *SET Alias\$AID HELP, this command produces the following:

```
Alias$AID : type String, value : HELP
```

Example: SHOW Alias\$*

Lists all aliases. If no parameter is given, all variables are listed.

***STATUS**

Syntax: *STATUS [<param>]

*STATUS displays a full list of the default values held in the CMOS RAM. If <param> is present, it must be one of the settings shown under *CONFIGURE and only the corresponding value is displayed.

Note that changes to the settings held in the battery backed RAM are only implemented after a power-on or hard break. *STATUS displays the settings which will come into effect when the machine is next switched on or subjected to a hard break. These may not be the same as the settings currently operating if *CONFIGURE has been used since they were last implemented.

***TIME**

Syntax: *TIME

*TIME produces a standard display containing the day, date and time. The format is held in the variable Sys\$DateFormat. The default value for the variable is:

```
%w3, %dy %m3 %ce%yr.%24:%mi:%se
```

It is converted into a string using OS_ConvertStandardDateAndTime. See this call for a description of the format string syntax. An example of a date printed using the default format is:

```
Mon, 29 Feb 1988.12:34:56
```

***TV**

Syntax: *TV [<0 - 255> [[,]<0 - 255>]]

*TV specifies the vertical screen alignment and interlace option. The first parameter is interpreted as a signed byte: 1 means a vertical adjustment of one line up, and 255 (-1) means a vertical adjustment of one line down. The second parameter controls the use of the screen interlace for future mode changes:

0	Switches interlace on
1	Switches interlace off

The default value for the first parameter, if it is omitted, is zero. If the second parameter is omitted, the current interlace setting is left unchanged. A *TV command doesn't come into effect until the next mode change.

***UNSET**

Syntax: *UNSET <[wildcarded] varname>

*UNSET may be used to delete any non-special variables which have been set up using *SET, etc. It deletes all those which match the wildcarded name given.

Example: *UNSET ALIAS\$*

Reading CLI parameters

If you are writing a module, the chances are that you will want to recognise one or more * commands. The chapter **MODULES** explains how you can cause the OS to recognise commands for you, and pass control to your module when one has been found. This section describes the OS calls which are available to facilitate the decoding of the rest of the command line.

The calls mentioned here may also be used by * commands activated in other ways, eg a transient command loaded from disc. However, the way in which the tail of the command line is discovered will vary for these types of commands. See the chapter **THE PROGRAM ENVIRONMENT** for details.

On entry to your * command routine, R0 contains a pointer to the 'tail' of the command, ie the first character after the command name itself (with spaces skipped). R1 contains the number of parameters, where a parameter is regarded as a sequence of characters separated by spaces.

The way in which the command uses the parameters depends on what it is doing. First, if there are too many or too few parameters, an error could be given. (A module can arrange for the OS to do this automatically.)

If a parameter is to be regarded as a string, OS_GSTrans may be used to decode any special sequences, eg control codes, variable names etc. If the parameter is a number, OS_ReadUnsigned might be used to convert it into binary. Finally, OS_EvaluateExpression could be used to read a whole arithmetic or string expression, and return the result in a buffer.

These calls are documented in the chapter **NUMBER CONVERSIONS**, along with other useful conversion routines such as OS_ReadUnsigned.

Note that the convention on the Archimedes is to have parameters separated by spaces. Some of the built-in commands which have been carried over from the BBC/Master machines also allow commas. You should not support this option.

INTRODUCTION

A filing system is a collection of routines provided for the purpose of organising and accessing data held on external storage media. Two complete filing systems are provided as standard:

- Advanced Disc Filing System (ADFS) which is for use with both floppy and hard disc drives.
- Network Filing System (NetFS) for controlling Econet file servers.

(The presence of the NetFS will only be apparent if the machine has Econet hardware installed.) The filing system which is selected when the computer is switched on is known as the default filing system and can be set using the *CONFIGURE File (or FileSystem – see the chapter **THE COMMAND LINE INTERPRETER**) command. To configure a filing system as default, type

```
*CONFIGURE File <n>
```

where <n> is the appropriate filing system number, 8 for ADFS, 5 for NetFS. The default configured filing system is ADFS; this is restored whenever a CMOS RAM reset is performed.

You can change the current filing system by issuing one of the following commands:

```
*ADFS    to select the Advanced Disc Filing System  
*NET     to select the Network Filing System
```

You can also switch filing systems for the duration of a * command by prefixing the command with the name of the filing system between minus signs:

```
*-net-cat
```

Programs cannot interact directly with the filing system. Instead, they communicate with the filing system manager (called FileSwitch). This layer of software performs tasks such as keeping track of which filing system an open file resides on, decoding

path names etc. It provides many of the filing system-independent commands, such as *CAT, *WIPE etc.

Additionally, the common filing system SWIs, such as OS_Find go through FileSwitch before being routed to the appropriate filing system module. The presence of FileSwitch has advantages for both the user and implementer of filing systems. For the user, it simplifies the task of remembering command names, as many of them are common to all filing systems. Also, FileSwitch allows files on filing systems other than the current one to be accessed. Switching between filing systems is done invisibly.

For the filing system writer, the presence of FileSwitch means that less code has to be written. This is because many high-level operations are performed by FileSwitch, which calls the appropriate lower-level filing system routines.

Some of the filing system commands, such as *LOAD and *SPOOL are provided by the OS kernel. These are listed under 'Utility commands' when you type a *HELP COMMANDS command. They are listed in this chapter along with the FileSwitch commands for completeness.

Another *CONFIGURE option is Boot / NoBoot. This determines whether the ADFS will try to access the file called !Boot when the machine is reset. If the configuration is set to Boot, the file will be dealt with according to *OPT 4 setting of the disc. See the *OPT command for more details.

The CMOS RAM default setting is NoBoot.

FILES, DIRECTORIES AND PATHNAMES

Under both the ADFS and NetFS all programs, data files, documents, etc are saved as files which are identified by filenames. The filing systems make no distinction between types of file. You are, therefore, able to load a word processor file with BASIC as the current language; it is only when the file has been loaded that the content of the file is found to be inconsistent with the format required by the language.

It is up to an application or utility to ensure that a file is of the desired type before loading it; for instance, the module handler will only allow files which are Relocatable Modules to be loaded into the module area.

Filenames may be up to ten characters in length on ADFS and NetFS. These characters may be digits or letters, with no distinction being made between upper and lower case. It is not advisable to use 8-bit characters in filenames. Other characters may be used provided they do not have a special significance. Those that do are listed below:

.	Separates directory specifications, eg \$.fred
:	Introduces a drive or disc specification, eg :0, :welcome it also marks the end of a filing system name, eg adfs:
*	Acts as a 'wildcard' to match zero or more characters, eg prog*
#	Acts as a 'wildcard' to match any single character, eg \$.ch##
\$	is the name of the root directory of the disc
&	is the user root directory (URD)
@	is the currently-selected directory (CSD)
^	is the 'parent' directory
%	is the currently-selected library directory (CSL)
}	is the previously-selected directory (PSD)

Files may be grouped together into directories. This is particularly useful for grouping together all files of a particular type. Files in the directory currently selected may be accessed without reference to the directory name. Filenames must be unique within a given directory. Directories may contain other directories, leading to a hierarchical file structure.

All files are accessible through the root directory, \$. This forms the top of the hierarchy. \$ does not have a parent directory. Trying to access its parent will just access \$.

Files in directories other than the current directory may be accessed either by making the desired directory the current directory, or by prefixing the filename by an appropriate directory specification. This is a sequence of directory names starting from one of the single-character directory names listed above, or from the current directory if none is given.

Each directory name is separated by a '.' character. For example:

\$.Documents .Memos	File Memos in dir Documents in \$
BASIC .Games .Adventures	File Adventures in dir Games in dir @.BASIC
% .BCPL	File BCPL in the current library

Files may also be accessed on filing systems other than the current one by prefixing the filename with a filing system specification. A filing system name may appear between '-' characters, or suffixed by a '.'. For example:

```
-net-$ .SystemMesg  
adfs:% .AAsm
```

In addition to the two filing systems already mentioned, the module 'System Devices' provides some device-oriented 'filing systems'. These can be used in redirection specifications in * commands, and anywhere else where byte-oriented file operations are possible. The devices provided are:

null	Output disappears; input gives EOF
vdu	Output goes to OS_WriteC; input is 'Not found'
rawvdu	Output goes to OS_WriteC; input is 'Not found'
printer	Output goes to printer; input is 'Not found'
kbd	Output is bad op; input comes from OS_ReadLine
rawkbd	Output is bad op; input comes from OS_ReadC

The difference between vdu and rawvdu is that the former is filtered in the same way as output from commands such as *TYPE (using OS_GSRead encoding on control characters etc), whereas rawvdu characters go straight to the VDU drivers. In addition to byte-oriented operations, you are allowed to perform file save operations on the output devices.

You can generate an EOF condition on the kbd device by typing **Ctrl**D before pressing **↵**. For example:

```
*COPY file vdu:
SAVE"VDU:"
*SPOOL printer:
LIST
*SPOOL
```

An error is given if the specified filing system/device is not present.

FILE TYPES AND DATE STAMPING

All files have, in addition to their name and length, two 32-bit fields describing them. These are set up when the file is created and have two possible meanings:

Load and execution addresses

In the case of a simple machine code program these are the load and execution addresses of the program:

Load address	XXXLLLLL
Execution address	GGGGGGGG

When a program is *RUN, it is loaded at address &XXXLLLLL and execution commences at address &GGGGGGGG. Note that the execution address must be within the program or an error is given. That is:

$$\text{XXXLLLLL} \leq \text{GGGGGGGG} < \text{XXXLLLLL} + \text{Length of file}$$

Also note that if the top twelve bits of the load address are all set (ie 'XXX' is FFF), then the file is assumed to be date-stamped. This is reasonable because such a load address is outside the addressing range of the ARM CPU.

Date/time stamp and file type

In this case the top 12 bits of the load address are all set. The remaining bits hold the date/time stamp indicating when the file was created or last modified, and the file type.

The date/time stamp is a five byte unsigned number which is the number of centi-seconds since 00:00:00 on 1st Jan 1900. The lower four bytes are stored in the execution address and the most-significant byte is stored in the least-significant byte of the load address.

The remaining 12 bits in the load address are used to store information about the file type. Hence the format of the two addresses is as follows:

Load address	FFFt t t d d
Execution address	d d d d d d d d

where 'd' is part of the date and 't' is part of the type.

The file types are split into three categories:

Value	Meaning
&E00 – &FFF	Reserved for Acorn use
&800 – &DFF	For allocation to software houses
&000 – &7FF	Free for the user

The types currently defined are:

Value	Type
&FFF	Plain ASCII text
&FFE	Command
&FFD	Data
&FFC	Position-independent transient – loaded and run in RMA
&FFB	Tokenised BASIC program
&FFA	Relocatable module
&FF9	Sprite
&FF8	Absolute code – runs as an application at &8000
&FF7	BBC font
&FF6	Fancy font

&FEF	Diary
&FEE	Note pad
&FED	Palette
&FE0	Desktop utility

When a date stamped file of type ttt is *LOADed or *RUN, the filing system manager looks for the variables Alias\$@LoadType_ttt or Alias\$@RunType_ttt respectively. If these exist, then the string LoadVal filename or RunVal filename parameters are passed to the operating system command line interpreter. LoadVal and RunVal are the values of the respective Alias\$ strings.

For example, suppose you type

```
*LOAD mySprites
```

where the type of the file mySprites is &FF9. The file manager will look up the value of the variable Alias\$@LoadType_FF9. This is SLoad %*0 by default, so the command actually passed on would be:

```
*SLoad mySprites
```

The filing system manager sets several of these variables up on initialisation, which you may override by setting new ones.

In the case of BASIC programs the settings are:

```
*SET Alias$@LoadType_FFB Basic -Load %*0
*SET Alias$@RunType_FFB Basic -Quit %*0
```

You can set up new aliases for any new types of file. For example, you could assign type &123 to files created by your own wordprocessor. The variables could then take the following form:

```
*SET Alias$@LoadType_123 WordProc %0
*SET Alias$@RunType_123 WordProc %0
```

- Note: in OS versions 0.40 and greater, the syntax of these variables is different from that described above. They now have the form Run\$Type_TTT and Load\$Type_TTT. The way in which they are used remains the same.

There are two more important variables used by FileSwitch. These control exactly where a file will be looked for, according to the operation being performed on it. The variables are:

File\$Path for read operations
Run\$Path for execute operations

The contents of each variable should be a list of directory names terminated with dots, separated by commas. When FileSwitch performs a read operation (eg load a file, open a file for input or update), then the directories in File\$Path are searched in the order in which they are listed.

Similarly, when FileSwitch tries to execute a file (*RUN, */ for example), the directories listed in Run\$Path are searched in order. By default, File\$Path is set to the null string, and the only directory searched is the current one. Run\$Path is set to '%.'. This means that the current directory is searched first, followed by the library.

It is possible to specify filing system names in the search paths. For example, if it can't locate a file on the ADFS you could make FileSwitch look on the fileservers using:

```
*SET File$Path ,%.,NET:LIB*.,NET:MODULES.
```

This would look for: @.file, %.file, NET:LIB*.file and NET:MODULES.file.

Note that the search paths in these two variables are only ever used when the pathname passed to FileSwitch does not contain an explicit directory reference. For example, *RUN file would use Run\$Path, but *RUN &.file wouldn't.

Certain calls also allow you to specify alternative path strings, and to perform the operation with no path look-up at all.

FILESWITCH AND OS FILING SYSTEM COMMANDS

The filing system commands given below are those which are common to all the filing systems. These commands are recognised and acted upon by the filing system manager or utility module, rather than by the individual filing systems themselves.

Where addresses or offsets are required in these commands, hexadecimal is taken as the default base. However, any other base may be specified by using the standard OS_ReadUnsigned notation. For example, 10_1000 means 1000 decimal.

***ACCESS**

Syntax: *ACCESS <[wildcarded]object> [D] [L] [W] [R] [/] [W] [R]

*ACCESS sets the file attributes according to the presence of absence of one or more of the optional parameters:

- 'L' locks objects so that they may not be deleted, written to or overwritten (for example, if a locked file is loaded, it may not be saved again with the same name).

Both files and directories may be locked. Note, however, that locking does not prevent directories or files from being destroyed if the disc is reformatted. Neither does it prevent modification of a directory.

- 'W' sets 'write access' (for files only). This must be set for writing to be allowed.
- 'R' sets 'read access' (for files only). This must be set for reading and loading to be allowed.

There is a further attribute, 'D', which is set if the object is a directory (see *CDIR below), and is unaffected by *ACCESS commands. Similarly, it is not possible to turn a file into a directory by attempting to set the 'D' attribute; the 'D' attribute is always ignored.

Attributes specified after the optional slash character relate to the access which other users may have to the file (on the Econet); these default to no access.

The default attributes (ie those which are assigned when an object is first created) are:

Object type	Attributes
Files	WR
Directories	DL

***APPEND**

Syntax: *APPEND <[wildcarded]filename>

*APPEND opens the named file, which must exist, and sets the file pointer to the end of the file. Subsequent lines of keyboard input will be appended to the end of the file. Input is terminated by the occurrence of an escape condition, which causes the file to be closed.

***BUILD**

Syntax: *BUILD <filename>

*BUILD opens a new file with the specified name and directs all subsequent lines of keyboard input to the file. Input is terminated by the occurrence of an escape condition, which causes the file to be closed. If an existing file of the same name already exists then it is overwritten, unless it is locked.

If the file already exists, and has a file type, then its type is unaltered, but the datestamp is updated. If the file didn't exist, or was unstamped, it is given type FFD (data). To make the file *EXECable, you should set its type to FFE.

***CAT**

Syntax: *CAT [<directory>]

*CAT displays a catalogue of the current directory or the specified directory.

***CDIR**

Syntax: *CDIR <directory> [<size in entries>]

*CDIR creates a new empty directory with the specified pathname.

The new directory is assigned DL attributes (see *ACCESS), and the directory name is also allocated as the directory title (see *TITLE under ADFS commands).

NetFS uses the size parameter to request a directory block sufficiently large to contain, at least, a given number of directory entries. ADFS ignores it.

***CLOSE**

Syntax: *CLOSE

*CLOSE closes all open files associated with the current filing system, after first ensuring that any modified data held in RAM is written out to the filing system.

***COPY**

Syntax: *COPY <[wildcarded]object> <[wildcarded]pathname> [options]

*COPY copies all files matching the specification of the source (first) parameter, to the place specified by the second parameter. If necessary, files may be copied from a Winchester disc to a floppy disc by including appropriate drive specification parameters. The file(s) are added to the catalogue for the specified directory and it is, therefore, possible to encounter `Dir full` or `Disc full` error messages.

If neither of the names contains wildcards, then a single file is copied. If these filenames are directories, then the contents of the source directory will be copied into the destination directory.

If the source contains a wildcard in the final component of the file name, then all files which match that will be copied. The destination should have an equivalent wildcard name. For example:

```
*COPY fred.* jim.*
```

copies all of the files in directory fred into jim. Note that wildcards used in directory names, in the source, only mean 'first match', rather than 'all matches'. Thus the command:

```
*COPY adfs:*.fred jim.file
```

means copy the file fred from the first directory found on the ADFS into jim.file.

The options are as follows:

- | | | |
|---|---------|---|
| C | Confirm | This causes a prompt for confirmation to be made before each file is copied. |
| D | Delete | This causes the source file to be deleted after it has been successfully copied. |
| F | Force | This causes any destination file to be overwritten if it already exists. |
| P | Prompt | This causes prompts to be made to insert the disc into the drive when copying between discs. It is useful for single-drive copies between discs. |
| Q | Quick | This causes the copy routine to use the application area (from &8000 to the current limit) as workspace if possible to speed up copying. It is dangerous in that you may lose your BASIC program, or even the current application if that was loaded at &8000. It is always safe to use this option from the supervisor prompt. A copy using application workspace will exit from the application when it has finished. |

R	Recurse	This causes any subdirectories of the source specification to be copied.
V	Verbose	This causes information to be displayed on each file being copied.

The default options are held in the variable `Copy$Options`. To cancel an option, precede it with '~'. For example:

```
*COPY * BAK. * ~R.
```

Note that it is dangerous to copy a directory into one of its subsidiary directories. This results in an infinite loop, which will only stop when the disk is full.

*CREATE

Syntax: *CREATE <filename> [<size> [<exec addr> [<load addr>]]]

*CREATE provides a means of reserving space for files. The amount of space reserved is the number of bytes given after the filename. If this is omitted, zero is used. The load and execution addresses to put on the file may be specified. If the execution address is given but the load address is omitted, the load address is set to zero.

If both execution and load addresses are omitted, then the file is created with type FFD (data) and is date/time stamped.

*DELETE

Syntax: *DELETE <object>

*DELETE deletes the specified object, if it exists, from the current file catalogue, so that it cannot be accessed, and the space it was occupying may be overwritten. If the file does not exist or is locked against deletion then an error message is given. Wildcards may be used in all components of the pathname except the last one.

*DIR

Syntax: *DIR [<dirname>]

*DIR changes the current directory. If no parameter is supplied then the current directory is reset to the user root directory. Where a parameter is given, the directory specified is made current. In either case, the current directory is saved (for use by *BACK) as the previously selected directory (PSD).

*DUMP

Syntax: *DUMP [<wildcarded>filename> [<file offset> [<start address>]]

*DUMP displays a hexadecimal and ASCII dump of the named file in the following format:

Address : 00 01 02 03 04 05 : ASCII data

XXXXXXXX : BB BB BB BB BB BB : ccccccc

The number of bytes displayed per line depends on the number of text columns in the current text window: one in 20-column modes, six in 40-column modes, 16 in 80-column modes, 24 in 132-column modes, and 32 in the 64 KHz-monitor modes.

The way in which the ASCII data part is displayed is determined by bits 2 and 3 of the *CONFIGURE DumpFormat value.

The file offset specifies the point in the file at which the dump is to start; this defaults to zero. The start address is used to determine the address printed on each line of the display, and is the address which would be displayed if the file offset were zero. If this is not present, then it defaults to the load address of the file unless the file is date/time stamped, in which case it is taken to be zero.

***ENUMDIR**

Syntax: *ENUMDIR <dirname> <filename> [<wildcard pattern>]

*ENUMDIR reads filename entries from the given directory and copies them into the specified file, if they match the wildcard pattern (the default being *). Entries are separated by ASCII 10 (line feed) in the output file. For example, the command *ENUMDIR \$ VDU: might produce the output:

```
Library
MAESTRO
TEST
TWIN
TWIN132
```

***EX**

Syntax: *EX [<[wildcarded]dirname>]

*EX displays file catalogue information for all objects in the current or specified directory. The information supplied is as follows:

filename	attributes or time	loadaddress or date	executionaddress or addr	size	start
----------	-----------------------	------------------------	-----------------------------	------	-------

All addresses are given in hexadecimal notation and sizes are in bytes.

***EXEC**

Syntax: *EXEC [<[wildcarded]filename>]

*EXEC opens the given filename for reading, and causes the characters from that file to be used for subsequent input. The file takes priority over the keyboard or RS423 input stream. This means that calls to OS_ReadC, OS_Byte &81, OS_Word &00, and OS_ReadLine will read from the file (as if by OS_BGet) instead of the selected input stream. When all the characters have been read, the file is closed automatically, and the input reverts to the previous source.

If no parameter is given to *EXEC then the current *EXEC file is closed. If another *EXEC command is issued with a filename parameter when there is already an *EXEC file active, the first file is closed and input is taken from the newly-opened one.

***INFO**

Syntax: *INFO <[wildcarded]object>

*INFO displays the same filing system information as *EX but for either a single object or a group of objects.

For example, *INFO font* will give information about all files beginning with font.

***LCAT**

Syntax: *LCAT [<[wildcarded]dirname>]

*LCAT displays the catalogue for the current library directory, or a subdirectory thereof, if a parameter is supplied. It is the same as *CAT %.

***LEX**

Syntax: *LEX [<[wildcarded]dirname>]

*LEX provides the same facility as *EX, but for the library directory or a subdirectory thereof, it does not require you to make the library directory current. It is the same as *EX %.

***LIB**

Syntax: *LIB [<[wildcarded]dirname>]

*LIB sets the library directory to the directory specified. If it is used with no parameters then the default library for the current filing system is selected.

Note that unlike *DIR, *LIB does not affect the current directory. It is usual to make one component of the Run\$Path variable % so that the current library is searched for programs which are *RUN.

*LIST

Syntax: *LIST <[wildcarded]filename>

*LIST displays the contents of the named file. The format used for control and 'international' characters depends on the setting of the *CONFIGURE DumpFormat value.

Each line (ie sequence of ASCII codes terminated by a carriage return, line feed or pairs of the above) is preceded with a line number, starting from 1.

*LOAD

Syntax: *LOAD <[wildcarded]filename> [<load address>]

*LOAD loads the specified file into memory. The address at which it is loaded can be specified as a hexadecimal (or other given base) value after the filename. Otherwise, the load address supplied by the filing system will be used. (See date/time section above for load actions of date/time stamped files.)

*OPT

Syntax: *OPT <option number> [<option value>]

*OPT sets up various filing system options:

*OPT 0 restores the default settings for all options associated with the current filing system, except *OPT4.

*OPT 1 controls the display of file information during load, save and create as follows:

- 0 File information is to be suppressed
- 1 The filename is to be displayed
- 2 The filename, load address, execution address and length are to be displayed
- 3 The filename and length are to be displayed as above, with the load and execution addresses being interpreted as file type and date/time stamp if possible

*OPT 4 controls the auto-start option for ADFS as follows:

- 0 Disable the auto-start facility
- 1 *LOAD the &.!BOOT file
- 2 *RUN the &.!BOOT file
- 3 *EXEC the &.!BOOT file

*OPT 4 controls the auto-start option for NetFS logon as follows:

- 0 Disable the auto-start facility
- 1 *LOAD the &.!ArmBoot file
- 2 *RUN the &.!ArmBoot file
- 3 *EXEC the &.!ArmBoot file

*PRINT

Syntax: *PRINT <[wildcarded]filename>

*PRINT displays the contents of the named file in ASCII format. Each byte is sent to the VDU driver regardless of whether it is a printable character or a control character. Hence, unless the file is a simple text file, this command can produce undesirable results. It may be used to 'replay' *SPOOLED graphics output, or a stream of VDU 19s to set-up the palette, for example.

***REMOVE**

Syntax: *REMOVE <object>

*REMOVE is identical to *DELETE with the exception that it does not give the error `Not found` if there is nothing to delete. This makes it most useful in a program since it will not cause an unexpected error.

***RENAME**

Syntax: *RENAME <object> <object>

*RENAME changes the pathname through which a file is accessed, from that specified by the first parameter to that specified by the second. The current directory is assumed if either or both directory specifications are omitted from the object(s).

Note that it is not possible to rename a locked object and that the object defined by the second parameter must not already exist. If the object is a directory, all files and subordinate directories remain unchanged but will be accessible only via their new pathname.

***RUN**

Syntax: *RUN <[wildcarded]filename> [<parameters>]

*RUN loads the named file into memory and starts execution. It uses the load and execution addresses stored by the filing system. The optional parameters may be accessed by the program itself. (See date/time section above for run actions of date/time stamped files.)

***SAVE**

Syntax: *SAVE <filename> <start> <end> [<exec> [<load>]]

or: *SAVE <filename> <start>+<length> [<exec> [<load>]]

*SAVE takes a copy of a designated area of memory and writes it to the named file.

start is the address of the first byte to be saved

end is the address of the byte after the last byte to be saved

length is the number of bytes to be saved

exec is the execution address to be stored with the file – this defaults to the start address

load is the reload address (which allows the load address stored with the file to be different from the actual start address used when saving the file, and which is assumed to be the same as start if omitted).

***SETTYPE**

Syntax: *SETTYPE <filename> <filetype>

This command sets the file type of the named file. If the filename contains wildcards, only the first file accessed by that name is affected. The file type is a number in the range 0 – &FFF. The default base is hexadecimal, but as usual, may be overridden by specifying the base.

***SHUT**

Syntax: *SHUT

*SHUT closes all open files, after first ensuring that any unwritten data remaining in RAM is written to the filing system. It is similar to *CLOSE except that it affects all filing systems, rather than just the current one.

***SHUTDOWN**

Syntax: *SHUTDOWN

This command performs all the actions of *SHUT. Additionally, it logs off all file servers and unmounts any ADFS discs (which in turn 'parks' the heads of any attached Winchester drives).

***SPOOL**

Syntax: *SPOOL [<filename>]

The command *SPOOL <filename> opens the specified file for output. All subsequent characters sent to the VDU drivers will also be copied to the file, using OS_BPut. This continues until the next *SPOOL command (with or without a file name) is issued.

If the filename is omitted, the current *SPOOL file, if any, is closed, and characters are no longer sent to it.

You can temporarily disable the *SPOOL file, without closing it, using the OS_Byte &03 call.

***SPOOLON**

Syntax: *SPOOLON [<[wildcarded]filename>]

*SPOOLON is similar to *SPOOL except that it takes the name of an existing file, to which all subsequent VDU output is appended. The file may be closed using either a *SPOOL or *SPOOLON command without a parameter.

***STAMP**

Syntax: *STAMP [<[wildcarded]filename>]

*STAMP sets the date/time stamp of an existing file to the current date/time of the computer. If the file is not already date/time stamped, then it is given file type FFD. This can be changed using *SETTYPE.

***TYPE**

Syntax: *TYPE [<[wildcarded]filename>]

*TYPE is similar to *LIST in that it displays the contents of the named file in the format dictated by the *CONFIGURE DumpFormat parameter. However, it does not precede each line with a line number.

*UP

Syntax: *UP [<number of levels>]

*UP without a parameter is equivalent to *DIR ^, ie it selects the parent of the current directory. When a parameter is given, it is taken to be the number of steps in the directory hierarchy that should be taken upwards, eg *UP 3 is equivalent to *DIR ^.^.^.

Note that the parent of '\$' is '\$', so you can't go any further up than this.

*WIPE

Syntax: *WIPE <[wildcarded]filename> [<options>]

This command deletes one or more files given by the (wildcarded) filename. If the filename given is a directory, the contents of that directory are deleted. The options are as follows:

- C Confirm. This causes a prompt for confirmation before each file is deleted.
- F Force. This causes objects to be deleted even if they are locked (eg directories).
- R Recurse. This causes the subdirectories of a directory to be deleted as well.
- V Verbose. This causes information about what is being deleted (and what isn't because of locking) to be printed.

The default options are held in the variable Wipe\$Options. To cancel an option, precede it with '~'. For example: *WIPE * ~R.

OS FILING SYSTEM SWI CALLS

The filing system SWIs given below are those which are common to all the filing systems. These commands are recognised and acted upon by FileSwitch, which then passes the call onto the appropriate filing system.

Pathname conventions

Pathnames passed to FileSwitch must terminate with a carriage return, line feed or NULL (ASCII 0) byte. Pathnames are OS_GSTRansformed before use, and so may contain references to variables in angled brackets, or may be enclosed within quotes, for example:

```
<device$prefix>.source.header
```

An error is given if the pathname is found to contain illegal character sequences.

In general, filenames, used in read operations, may contain wildcards. Thus, a filename being opened for input, or loaded into memory, may be wildcarded. The first name which matches will be used. The last component of a filename used in save or delete-type operations must not contain wildcards. Opening a file for output implies deletion (of any file with the same name), so a name given in one of these operations may not be wildcarded.

In addition to the specialised filing system calls listed from OS_File onwards, two OS_Bytes are provided for filing system control:

OS_Byte &7F (127) – Check for end of file

On entry: R1 = file handle

On exit: R1 indicates if end of file has been reached

This call enables you to ascertain whether the end of an open file has been reached. See OS_Find below for details of opening a file. The values returned in R1 are as follows:

Value	Meaning
0	End of file has not been reached
Not 0	End of file has been reached

OS_Byte &8B (139) – Write filing system options

On entry: R1 = option number (first *OPT argument)
R2 = option value (second *OPT argument)

On exit: R1 is undefined
R2 is undefined

This call selects file options. It is equivalent to *OPT which is documented in detail in the next section **FileSwitch and OS filing system commands**.

In addition to these, there are OS_Bytes to read/write the *SPOOL and *EXEC file handles. See the chapters **CHARACTER OUTPUT** and **CHARACTER INPUT** respectively for details.

OS_Byte &FF (255) – Read/write boot option

On entry: R1 = 0 or new value
R2 = &FF or 0

On exit: R1 previous value

This call reads the current auto-boot flag setting, or temporarily changes it. The auto-boot flag defaults to the value configured in the Boot/NoBoot option. If NoBoot is set, then, when the machine is reset, no auto-boot action will occur (ie no attempt will be made to access the !Boot file on the filing system). If Boot is the configured option, then !Boot will be accessed on reset. Either way, holding down the **Shift** key while releasing RESET will have the opposite effect to usual.

With this OS_Byte you can read the current state. On exit, if R1=0 then the action is Boot. If it is 8, then the action is NoBoot. The effect can be changed by writing to the flag, but this only lasts until the next hard reset.

OS_File (&08) – Perform action on whole file

OS_File acts on whole files, either loading a file into memory, saving a file from memory, or reading or writing a file's attributes. The call indirects through FileV.

For calls with R0=&05 (read catalogue) and &FF (load file), the file is searched for using the variable File\$Path. If this does not exist, a null path string is used (ie look only in current directory).

Some operations do not allow wildcard characters (* and #) to be used in the filename. These are the ones which have a 'destructive' effect, eg deleting a file or saving a file (which might overwrite a file which already exists). Non-destructive operations, such as loading a file and reading and writing attributes may have wildcards in the filename. However, only the first file found (in ASCII order of file name) will be accessed by the operation.

The particular action of OS_File is specified by R0 as follows:

R0 = 0 Save a block of memory as a file

On entry: R1 = pointer to non-wildcarded filename
 R2 = reload address of file
 R3 = execution address of file
 R4 = start address in memory of data
 R5 = end address in memory of data

On exit: –

An error is given if the object is locked against deletion, or if it is a directory, or is already open.

R0 = &01 Write catalogue information for the named file

On entry: R0 = action
R1 = pointer to (wildcarded) filename
R2 = load address of file
R3 = execution address of file
R5 = file attributes

On exit: –

The load address, execution address and file attributes from registers R2, R3 and R5 are written to the named file's catalogue entry. An error is given if the object is a directory. It is not an error if the file does not exist.

If the filename contains wildcards, only the first file matching the wildcard specification will be affected.

R0 = &02 Write load address only for the named file

On entry: R1 = pointer to (wildcarded) filename
R2 = new load address of file

On exit: –

An error is given if the object is a directory. It is not an error if the file does not exist

If the filename contains wildcards, only the first file matching the wildcard specification will be affected.

R0 = &03 Write execution address only for the named file

On entry: R1 = pointer to (wildcarded) filename
R3 = new execution address of file

On exit: –

An error is given if the object is a directory. It is not an error if the file does not exist.

If the filename contains wildcards, only the first file matching the wildcard specification will be affected.

R0 = &O4 Write attributes only for the named object

On entry: R1 = pointer to (wildcarded) pathname
R5 = new file attributes

On exit: --

It is not an error if the file does not exist or is a directory.

If the filename contains wildcards, only the first file matching the wildcard specification will be affected.

R0 = &O5 Read catalogue information for the named object

On entry: R1 = pointer to (wildcarded) pathname

On exit: R0 = object type
R1 is preserved
R2 = load address
R3 = execution address
R4 = file length
R5 = file attributes

The load address, execution address, length and file attributes from the named object's catalogue entry are read into registers R2, R3, R4 and R5. On exit, R0 contains the object type:

Value	Type
0	Not found
1	File found
2	Directory found

The top 24 bits of the file attributes are filing system dependent, eg NetFS returns the file server date of creation/modification of the object. The first byte has the following interpretation:

Bit	Meaning if set
0	Object has read access for you
1	Object has write access for you
2	Undefined
3	Object is locked against deletion
4	Object has read access for others
5	Object has write access for others
6	Undefined
7	Undefined

On ADFS, bit 4 has the same value as bit 0, and bit 5 is the same as bit 1. In calls to write attributes on ADFS, only bits 0, 1 and 3 are significant.

R0 = 006 Delete the named object

On entry: R1 = pointer to non-wildcarded pathname

On exit: R0 = object type
R1 is preserved
R2 = load address
R3 = execution address
R4 = file length
R5 = file attributes

The information in the named object's catalogue entry is transferred to the registers and the object is then deleted from the catalogue. It is not an error if the object does not exist.

An error is given if the object is locked against deletion, or if it is a directory which is not empty, or is already open.

R0 = &07 Create an empty file

On entry: R1 = pointer to non-wildcarded filename
 R2 = reload address of file
 R3 = execution address of file
 R4 = start address
 R5 = end address

On exit: —

The size of the empty file is determined by the start address and end address given in R4 and R5, but no data is transferred. It is usually convenient to set the start address to zero and use the end address to define the length of the file.

— *Note:* a file thus created does not necessarily contain zeros; the contents may be completely random.

An error is given if the object is locked against deletion, or if it is a directory which is not empty, or is already open.

R0 = &08 Create a directory

On entry: R1 = pointer to non-wildcarded directory name
 R4 = number of entries (0 for default)

On exit: —

R4 indicates a minimum number of entries that the created directory may contain. Zero is used to set the default number of entries.

– *Note:* ADFS ignores the number of entries parameter as this is predetermined by the disc format.

An error is given if the object is a file which is locked against deletion. It is not an error if it refers to a directory that already exists, in which case the operation is ignored.

R0 = &09 Write date/time stamp of file

On entry: R1 = pointer to (wildcarded) filename

On exit: –

The named file is date/time stamped using the current time. If the file was already date/time stamped, its file type is preserved, otherwise it is given file type &FFD.

An error is given if the object is a directory. It is not an error if the file does not exist.

R0 = &0A Save a block of memory as a file with current date/time stamp

On entry: R1 = pointer to non-wildcarded filename

R2 = file type

R4 = start address in memory of data

R5 = end address in memory of data

On exit: –

This operation is the same as the simple save operation except that the load and execution addresses of the file are written by the filing system manager to be a date/time stamped file of the given type with the current date/time stamp.

An error is given if the object is locked against deletion, or if it is a directory which is not empty, or is already open.

R0 = &0B Create an empty file with current date/time stamp

On entry: R1 = pointer to non-wildcarded filename
 R2 = file type (of form &0000TTTT)
 R4 = start address
 R5 = end address

On exit: –

This operation is the same as the simple create operation except that the load and execution addresses of the file are written by the filing system manager to be a date/time stamped file of the given type with the current date/time stamp.

An error is given if the object is locked against deletion, or is a directory which is not empty, or is already open.

R0 = &0C Load file with path string

On entry: R1 – R3 = as OS_File with R0 = &FF
 R4 = pointer to a path string

On exit: R0 – R5 = as OS_File with R0 = &FF

This call loads a file into memory, using a specified path string to look for the file. This is used instead of the default path string which is held in the variable File\$Path. The string is control character-terminated.

R0 = &0D Read catalogue info of file using given path string

On entry: R1 = pointer to (wildcarded) pathname
 R4 = pointer to path string

On exit: R0 – R5 = as OS_File with R0 = &05

This call reads a file's catalogue information. It uses the path specified in the string pointed to by R4 instead of the default search path stored in the variable Load\$Path.

R0 = §0E Load file with path variable

On entry: R1 – R3 = as OS_File with R0 = &FF
R4 = pointer to a path variable

On exit: R0 – R5 = as OS_File with R0 = &FF

This call loads a file into memory, using a path string which is held in the specified variable name to look for the file. This is used instead of the default path string which is held in the variable File\$Path. The variable name is control character-terminated.

R0 = §0F Read catalogue info of file using given path variable

On entry: R1 = pointer to (wildcarded) pathname
R4 = pointer to path variable

On exit: R0 – R5 = as OS_File with R0 = &05

This call reads a file's catalogue information. It uses the path specified in the contents of the string variable pointed to by R4 instead of the default search path stored in the variable File\$Path.

R0 = §10 Load file using no path at all

On entry: R1 – R3 = as OS_File with R0 = &FF

On exit: R0 – R5 = as OS_File with R0 = &FF

This call loads a file into memory, using just the path name specified by R1 on entry to find the file. No prefixes of any kind are prepended to the pathname given.

R0 = §11 Read catalogue info of file using no path at all

On entry: R1 = pointer to (wildcarded) pathname

On exit: R0 – R5 = as OS_File with R0 = &05

This call reads a file's catalogue information. No search path string is used. Only the pathname pointed to by R1 will be tried.

R0 = *012* Set file type

On entry: R1 = pointer to path name to set
R2 = file type (bits 0 – 11)

On exit: –

This call sets the file type of the specified file. If the file is not already date/time-stamped, it is set to the current time. If the file already has a date stamp, this is unaltered.

R0 = *0FF* Load the named file into memory

On entry: R1 = pointer to filename
R2 = load address of file (if R3b=0)
R3 = load at own / load at given flag

On exit: R0 = 1 (object is a file)
R1 is preserved
R2 = load address
R3 = execution address
R4 = file length
R5 = file attributes

The named file is loaded into memory at a location determined by the contents of R3:

- If the least significant byte of R3 is zero, the file is loaded into memory at the address specified in R2.
- If the least significant byte of R3 is non-zero, the file is loaded into memory using the file's own load address.

An error is given if the file does not exist, or is a directory, or does not have read access.

OS_Find (&OD) – Open or close a file for byte access

OS_Find opens and closes files. Opening a file declares a file requiring byte access to the filing system. Closing a file declares that byte access is complete. To use OS_Args, OS_BGet, OS_BPut or OS_GBPB with a file, it must first be opened. When used to open a file, OS_Find returns a 'handle', which is a unique identifier through which the file contents are made available to applications. This handle must always be passed to further OS calls in order to reference the file.

This call indirections through FindV

On entry: R0 = action
R1 = file handle to close (if R0 = 0)
R1 = pointer to pathname (if R0 <> 0)
R2 = pointer to path (optional, for open in and update)

On exit: R0 = file handle, or is preserved if R0=0 on entry

The particular action is determined by the top two bits of R0 as follows:

R0 = \$00 Indicates that an open file is to be closed

If R1 is zero then all files which are currently open and associated with the current filing system are closed.

If R1 is non-zero, it is taken to be a file handle. The corresponding open file is closed, after any modified data in RAM buffers has been written out (to the disc).

R0 not 0 Indicates that a file is to be opened

R1 points to the location in memory containing the first character of the pathname.

$R0 = \text{\$}4X (64+n)$ Indicates that a file is to be opened for input

The pathname may contain wildcards. If the file does not exist, then a file handle of zero is returned in $R0$; V will be clear, so this is not an error. Otherwise the file is opened for reading only, and a unique file handle passed back to the caller in $R0$.

The file is searched for using the contents of variable `File$Path`. If this does not exist, a null path string is used (ie looks only in current directory). In fact, the default setting of `File$Path` is the null string anyway. If you want to search the current directory, then you might use the library:

```
*SET File$Path , %.
```

$R0 = \text{\$}8X (128+n)$ Indicates that a file is to be created and opened for update

If the named file already exists, it will be opened for update and its extent (and file pointer) set to zero. If the file does not exist, a new file is created, which is then opened. It is an error if the object is a directory, or is locked against deletion. The filename may not contain wildcards.

$R0 = \text{\$}CX (192+n)$ Indicates that a file is to be opened for update

If the file does not exist, then a file handle of zero is returned in $R0$; this is not an error. Otherwise the file is opened for update, and a unique file handle passed back to the caller in $R0$. The filename obeys the same rules as `OS_Find &40`.

The bottom four bits of $R0$ indicate how the file will be opened. The bottom two bits control where the file be looked for, ie which search path will be followed, as set out below:

Bit 1	Bit 0	Meaning
0	0	Use the contents of <code>File\$Path</code> as the search path
0	1	Use the path string pointed to by <code>R2</code>
1	0	Use the path variable whose name is pointed to by <code>R2</code>
1	1	Use no path at all.

If bit 2 is set, then an error is given if an attempt is made to open a directory. If it is clear, you can open a directory, but no operations can be performed on it.

If bit 3 is set, then open for input and update calls will return an error if the file wasn't found (instead of a zero handle). This obviates the necessity of making explicit checks on the file handle.

OS_GBPB (&0C) – Read/write a group of bytes from/to an open file

OS_GBPB transfers a number of bytes to or from an open file.

This call indirects through GBPBV.

On entry: R0 = action
R1 – R6 depend on R0

On exit: R0 is preserved
R1 is preserved
R2 – R4 may be updated, depending on action

The particular action of OS_GBPB is determined by R0 as follows:

R0 = 1 Write bytes to an open file using specified file pointer

On entry: R1 = file handle
R2 = start address of data in memory
R3 = number of bytes to write to file
R4 = sequential file pointer to use for start of block

On exit: R1 is preserved
R2 = memory address of byte after the last one written
R3 = 0
R4 = initial R4 + initial R3 = pointer to next byte in file
C flag is clear

Data is transferred from memory to the file at the specified file pointer. If this is beyond the end of the file, the file is extended (with zeros) before the bytes are transferred.

The memory pointer is incremented for each byte written, and the final value is returned in R2. The sequential pointer of the file is incremented for each byte written, and the final value is returned in R4.

If the next read from the file produces an `End of file` error, this condition is cancelled, though EOF will still be true. The condition that a read from the file would produce an EOF error is referred to as EOF-error-on-next-read.

An error is given if the file is a directory, or does not have write access.

R0 = 2 Write bytes to an open file using file pointer

On entry: R1 = file handle
 R2 = memory address to take data from
 R3 = number of bytes to write to file

On exit: R1 is preserved
 R2 = memory address of byte after the last one written
 R3 = 0
 R4 = original sequential pointer + initial R3
 C flag is clear

Data is taken from memory and written to the file at the current sequential file pointer. The memory pointer is incremented for each byte written, and the final value is returned in R2. The sequential pointer is incremented for each byte written, and the final value is returned in R4. The EOF-error-on-next-read flag is cleared.

An error is given if the file is a directory, or does not have write access.

R0 = 3 Read bytes from a specified position in a file

On entry: R1 = file handle
R2 = start address of data in memory
R3 = number of bytes to read from file
R4 = sequential file pointer to use for start of block

On exit: R1 is preserved
R2 = memory address of byte after the last one read
R3 = number of bytes not read
R4 = initial R4 plus number of bytes transferred
C flag is clear if R3=0, else set

Data is transferred from the given file to memory using the specified file pointer and memory address. If the file pointer is greater than the current file extent then no bytes are read, and the sequential file pointer is not updated. Otherwise the sequential file pointer is set to the specified file location.

The memory address is incremented for each byte read, and the final value is returned in R2. The sequential pointer is incremented for each byte read, and the final value is returned in R4. The EOF-error-on-next-read flag is cleared.

If R3 is zero on exit (all the bytes were read), the carry flag will be clear, otherwise it is set.

An error is given if the file is a directory, or does not have read access.

R0 = 4 Read bytes from the current position in the file

On entry: R1 = file handle
R2 = start address of data in memory
R3 = number of bytes to read from file

On exit: R1 is preserved
 R2 = memory address of byte after the last one read
 R3 = number of bytes not read
 R4 = original sequential pointer plus number of bytes transferred
 C flag is clear if R3=0, else set

Data is transferred from the given file to memory using the current file pointer and the given memory address. The memory pointer is incremented for each byte read, and the final value is returned in R2. The sequential pointer is incremented for each byte read, and the final value is returned in R4. The EOF-error-on-next-read flag is cleared.

If R3 is zero on exit (all the bytes were read), the carry flag will be clear, otherwise it is set.

An error is given if the file is a directory, or does not have read access.

R0 = 5 *Read name and boot (*OPT 4) option of disc*

On entry: R2 = start address of data in memory

On exit: C flag is undefined

This call obtains the name of the disc which contains the current directory, and its boot option. This data is returned in the area of memory pointed to by R2, in the following format:

<name length byte><disc name><boot option byte>

R0 = 6 *Read current directory name and privilege byte*

On entry: R2 = start address of data in memory

On exit: C flag is undefined

This call obtains the name of the currently selected directory, and privilege status in relation to that directory. This data is returned in the area of memory pointed to by R2, in the following format:

<zero byte><name length byte><current directory name><privilege byte>

The directory name may contain trailing spaces.

The privilege byte is &00 if the user has 'owner' status (ie can create and delete objects in the directory) or &FF if the user has 'public' status (ie is prevented from creating and deleting objects in the directory). On ADFS the user always has owner status.

R0 = 7 Read library directory name and privilege byte

On entry: R2 = start address of data in memory

On exit: R2 is preserved
C flag is undefined

This call obtains the name of the library directory, and privilege status in relation to that directory. This data is returned in the area of memory pointed to by R2, in the following format:

<zero byte><name length byte><library directory name><privilege byte>

The directory name may contain trailing spaces.

R0 = 8 Read entries from the current directory

On entry: R2 = start address of data in memory
R3 = number of object names to read from directory
R4 = start offset in directory

On exit: R2 is preserved
 R3 = number of filenames not read
 R4 = next offset in directory
 C flag is clear if R3=0, else set

R3 contains the number of filenames to read. R4 is the offset in the directory to start reading (ie if it is zero, the first item read will be the first file). Filenames are returned in the area of memory specified in R2. The format of the returned data is:

length of first filename (one byte)
 first filename in ASCII (length as specified)

... repeated as specified by R3 ...

length of last filename (one byte)
 last filename in ASCII (length as specified)

If R3 is zero on exit, the carry flag will be cleared, otherwise it will be set.

R0 = 9 Read entries from specified directory

On entry: R1 = pointer to directory name (null terminated)
 R2 = start address of data in memory
 R3 = number of objects to read
 R4 = offset of first item to read in directory
 R5 = buffer length
 R6 = pointer to (wildcarded) name to match

On exit: R3 = number of objects read
 R4 = offset of next item to read (-1 if finished)
 C flag is clear if R3=0, else set

This call reads the names of entries in a directory into an area of memory pointed to by R2. If the directory name (which may contain wildcards) is null (ie R1 points to a zero byte), then the currently-selected directory is read.

The names which match the wildcard name pointed to by R6 are returned in the buffer as a list of null terminated strings, and R3 indicates how many were read. R4 contains the value which should be used on the next call (to read more names), or -1 if there are no more names after the ones read by this call.

Note that even if R3 returns with 0, the buffer area may still have been overwritten: it will contain filenames which did not match the wildcard name pointed to by R6.

R0 = 10 Read directory entries and information

On entry: R1 = pointer to directory name (null terminated)
R2 = start address of data in memory
R3 = number of object names to read
R4 = offset of first item to read in directory
R5 = buffer length
R6 = pointer to (wildcarded) name to match

On exit: R3 = number of records read
R4 = offset of next item to read (-1 if finished)
C flag is undefined

This call reads the names of entries in the given directory into memory pointed to by R2. If the directory name is null, then the currently-selected directory is read. The names and information are returned in records, with the following format:

Offset	Contents
&00	Load address
&04	Execution address
&08	Length
&0C	Attributes
&10	Object type
&14	Object name (null terminated)

Each record is word-aligned. For the meanings of the fields, see OS_File.

OS_BPut (&OB) – Write single byte to an open file

OS_BPut writes the byte given in R0 to the specified file at the current sequential file pointer. The sequential pointer is then incremented, and the EOF-error-on-next-read flag is cleared.

This call indirections through BPutV.

On entry: R0 = byte to be written
R1 = file handle

On exit: –

An error is given if the file is a directory, is not open for update, or does not have write access.

OS_BGet (&OA) – Read single byte from an open file

OS_BGet returns the byte at the current sequential file pointer position. If the EOF-error-on-next-read flag is set on entry, then an `End of file` error is given. If the sequential pointer is equal to the file extent (ie trying to read at end-of-file) then the EOF-error-on-next read flag is set, and the call returns with the carry flag set, R0 being undefined. Otherwise, the sequential file pointer is incremented and the call returns with the carry flag clear.

This mechanism allows one attempt to read past the end of the file before an error is generated. Note that various other calls (such as OS_BPut) clear the EOF-error-on-next-read flag.

This call indirections through BGetV.

On entry: R1 = file handle

On exit: R0 = byte read if C clear, undefined if C set

An error is given if the file is a directory, or does not have read access.

OS_Args (&09) – Read or write arguments for an open file

OS_Args reads or writes an open file's arguments or returns the filing system type in use.

This call indirects though ArgsV.

On entry: R0 = action
R1 = file handle or 0
R2 = attribute to write or not used

On exit: R0 = filing system number or is preserved
R1 is preserved
R2 = attribute that was read or preserved

The particular action of OS_Args is specified by R0 as follows:

R0 = 0 Read pointer/FS number

R1 = 0 Return currently-selected filing system number in R0
R1 = file handle Return sequential pointer for file in R2

R0 = 1 Write pointer

R1 = file handle Write sequential pointer for file from R2

If the new sequential pointer is greater than the current extent, then more space is reserved for the file; this new space will be filled with zeros. Writing the sequential pointer clears the EOF-error-on-next-read flag for this file.

R0 = 2 Read extent

R1 = file handle Return extent (ie length) of file in R2

R0 = 3 Write Extent

R1 = file handle Write extent of file from R2

If the new extent is greater than the current extent, then more space is to be reserved for the file; this new space will be filled with zeros. If the new extent is less than the current sequential pointer, then the sequential pointer will be set back to the new extent. Writing the extent clears the EOF-error-on-next-read flag for this file.

R0 = 4 Read allocated size

R1 = file handle Return size allocated to file in R2

The size allocated to a file will be at least as big as the current file extent; in many cases it will be larger. This call determines how many more bytes can be written to the file before the filing system has to allocate more space to the file.

R0 = 5 Read EOF status

R1 = file handle Return end-of-file indication

If the sequential pointer is equal to the extent of the given file, then an end-of-file indication is given, with R2 set to non-zero on exit. Otherwise R2 is set to zero on exit.

R0 = 6 Reserve space

R1 = file handle Ensure file size of at least R2 bytes

The filing system is instructed to ensure that the size allocated for the given file is at least that requested. Note that this space thus allocated is not yet part of the file, so the extent is unaltered, and no data is written. R2 on exit indicates how much space the filing system actually allocated.

RO = &FF Ensure file/files

R1 = 0 Ensure that any buffered data has been written to all files open on the current filing system

R1 = file handle Ensure that any buffered data has been written to the specified file

OS_FSCControl (&29) – Filing system control

OS_FSCControl controls the filing system manager and filing systems. This call indirects through FSCV and the particular action is determined by RO as follows:

RO = 0 Set current directory

On entry: R1 = pointer to wildcarded directory name

On exit: –

This call sets the current directory to the one identified by the name given. If the name is null, the directory is set to the filing system default (typically the same as the user root directory).

RO = 1 Set library directory

On entry: R1 = pointer to wildcarded directory name

On exit: –

This call sets the current library directory to the one identified by the name given. If the name is null, the library directory is set to the filing system default (typically \$.Library, if present).

RO = 2, 3

These calls are reserved.

RO = 4 Run file

On entry: R1 = pointer to wildcarded filename

On exit: -

This call runs a file, either as an absolute application (if not time/type stamped) or by using the corresponding *Alias\$@RunType* set up for the given file type, eg *EXECing command files (which will return), *RMRunning relocatable modules etc.

Transient code modules (type &FFC) are loaded into the RMA and executed there. Transient calls are nestable; when a transient returns to the filing system manager the RMA space is freed. The RMA space is also freed (on the reset service or filing system manager death) if the transient execution stopped abnormally, eg an exception occurred or RESET was pressed.

See the chapter **THE PROGRAM ENVIRONMENT** for details on writing transient utilities.

The file is searched for using the variable *Run\$Path*. If this does not exist, a path string of ',%.' is used (ie looks first in current directory, then in the library directory).

An error is given if the file does not exist, or is a directory, or does not have read access, or is a date/time stamped file without a corresponding alias set up for the given file type.

RO = 5 Catalogue directory

On entry: R1 = pointer to wildcarded directory name

On exit: -

This call catalogues the directory identified by the name given. If the name is null, the current directory is catalogued.

R0 = 6 Examine current directory

On entry: R1 = pointer to wildcarded directory name

On exit: -

This call prints information on all the objects in the directory identified by the name given. If the name is null, the current directory is examined.

R0 = 7 Catalogue library directory

On entry: R1 = pointer to wildcarded directory name

On exit: -

This call catalogues the specified subdirectory relative to the current library directory. If the name is null, the current library directory is catalogued.

R0 = 8 Examine library directory

On entry: R1 = pointer to wildcarded directory name

On exit: -

This call prints information on all the objects in the specified subdirectory relative to the current library directory. If the name is null, the current library directory is examined.

R0 = 9 Examine object(s)

On entry: R1 = pointer to wildcarded pathname

On exit: -

This call prints information on all the objects matching the wildcarded pathname given, in the same format as for Examine directory.

R0 = 10 Set filing system options

On entry: R1 = option
R2 = parameter

On exit: -

This call sets filing system options. An option of 0 means reset all filing system options to their default values. See the *OPT command.

R0 = 11 Set filing system from named prefix

On entry: R1 = pointer to string

On exit: R1 points past the filing system specifier if present
R2 = -1, no filing system was specified
R3 = pointer to special field O or 0

This call sets the filing system from a filing system prefix at the start of the string if one is present.

R0 = 12 Add filing system

This call is described in the section **Writing your own filing system.**

R0 = 13 Lookup filing system

On entry: R1 = filing system number or name
R2 = depends on R1

On exit: R1 = filing system number
R2 = pointer to filing system control block or 0 if not found

This call can be used to check for the presence of a filing system. If R1 is less than 256 then it points to the filing system number; if, however, it is over 255 then it points to the filing system name. If R2 is 0, the filing system name is taken to be

terminated with any control character or the characters: '#', ':' or '-'. If R2 is not 0, then the filing system name is terminated by any control character.

R0 = 14 Filing system selection

This call is described in the section **Writing your own filing system**.

R0 = 15 Boot filing system

On entry: -

On exit: -

This call is used by the operating system after **Shift Break**. The call uses the currently-selected filing system.

R0 = 16 Filing system removal

This call is described in the section **Writing your own filing system**.

R0 = 17 Add secondary module

This call is described in the section **Writing your own filing system**.

R0 = 18 Decode file type into text

On entry: R2 = file type

On exit: R2, R3 = textual form of file in registers

This call issues a look-up file type service(&42 or decimal 66). If the service is unclaimed, then it builds a default file type. For example if the file type is:

TEXT

the call returns:

&TEXT

The string is padded on the right to a maximum of 8.

R0 = 19 Restore current filing system

This call forces the temporary filing system to become the current one.

On entry: -

On exit: -

R0 = 20

This is reserved for operating system use.

R0 = 21 Returns filing system handle

This call returns the filing system handle associated with the file manager handle.

On entry: R1 = file handle

On exit: R1 = filing system handle
R2 = filing system information word

If the file manager handle is invalid, then it returns 0 as the filing system handle.

R0 = 22 Shut

Closes all open files, after first ensuring that any unwritten data remaining in RAM is written to the filing system. You can either use this call or the command *SHUT, the effect is identical.

On entry: -

On exit: -

R0 = 23 Shutdown

This call performs the actions of Shut as well as logging off all file servers and unmounting any ADFS discs. You can either use this call or the command *SHUTDOWN, the effect is identical.

On entry: --

On exit: --

R0 = 24 Set attributes of object(s)

On entry: R1 = pointer to wildcarded pathname
R2 = pointer to attribute string

On exit: -

This call gives the requested access to all objects matching the wildcarded name given. This call is used by the command *ACCESS.

R0 = 25 Rename object(s)

On entry: R1 = pointer to first pathname
R2 = pointer to second pathname

On exit: --

This call renames an object. It is a 'simple' rename, implying that the source and destination are single files which must reside on the same physical device. This call is used by the command *RENAME.

R0 = 26 Copy object(s)

On entry: R1 = pointer to first wildcarded pathname
 R2 = pointer to second wildcarded pathname
 R3 = mask describing the action
 R4 = optional start time
 R6 = optional start time
 R7 = optional end time
 R8 = optional end time

On exit: -

This call copies an object, optionally recursing. The filing system manager performs the copy.

The action mask contains 9 bits. These can be set in the following ways:

- Bit 8 If set, this allows printing during copy. Printing is otherwise disabled.
- Bit 7 If set, this deletes the source after copy. You can use this delete option to rename files across media.
- Bit 6 If set, this prompts the user to change media during the copy operation.
- Bit 5 If set, copy uses application workspace as well as the relocatable module area. This is, in fact, equivalent to the 'Q' option of the *COPY command.
- Bit 4 If set, maximum information is printed during copy.
- Bit 3 If set, this displays a Yes/No message prompting the user for each object to be copied.
- Bit 2 If set, this copies only files with a time/date stamp falling between the specified start and end time/date. Unstamped files will also be copied.

Bit 1 If set, this automatically unlocks and overwrites the specified file. No warning message is given.

Bit 0 If set, this allows recursive copying down directories.

R0 = 27 *Wipe object(s)*

On entry: R1 = pointer to wildcarded pathname to delete
R2 = not used
R3 = mask describing the action
R4 = optional start time
R6 = optional start time
R7 = optional end time

On exit: -

This call is used to delete files. You can modify the effect of the call with the action mask in R3. The function of the 9 bits is as for the Copy object(s) call above.

R0 = 28 *Count object(s)*

On entry: R1 = pointer to wildcarded pathname to count
R2 = not used
R3 = mask describing the action
R4 = optional start time
R6 = optional start time
R7 = optional end time

On exit: R2 = total number of bytes of all files that were counted
R3 = number of files counted

You can use this call to obtain information on the number and size of files. You can modify the effect of the call with R3, the action mask. The 9 bits are described in the Copy object(s) call above.

R0 = 29, 30

These calls are reserved for operating system use.

THE ADVANCED DISC FILING SYSTEM

The Advanced Disc Filing System (ADFS) is for use on both Winchester hard disc drives and 80 track double-sided floppy disc drives. On the floppy drives the following recording formats are available:

Format	Tracks	Density	Sectors/track	Bytes/sector	Storage
L	80	Double	16	256	640 Kbytes
D	80	Double	5	1024	800 Kbytes

Using the L format you can create 47 entries in each directory. Top-bit-set characters are not allowed in pathnames; using the D format, 77 entries may be created, and top-bit-set characters are allowed in pathnames.

Files stored on ADFS are sequences of bytes which always begin at the start of a sector and extend for the number of sectors necessary to accommodate the data contained in the file. The last sector used to accommodate the file may have a number of unused bytes at the end of it. The last 'data' byte in the file is derived from the file length stored in the catalogue entry for the file, or if the file is open, from its extent.

When ADFS is initialised on **Ctrl**RESET or power on, CMOS RAM is interrogated to find the values of configured information pertaining to ADFS. This comprises the following information:

Dir / NoDir

This determines whether ADFS will access information from the configured drive on initialisation. If Dir is configured, then the root directory of the disc in the configured drive is made the currently selected directory. If a directory called \$.Library exists on that disc, it is made the library directory, otherwise the library directory remains "Unset". The user root directory always starts "Unset".

If NoDir is configured, then the disc is not accessed, and all three main directories are made "Unset".

To set this, type *CONFIGURE Dir or *CONFIGURE NoDir (CMOS default is NoDir)

Drive <n>

This is the drive that ADFS will use by default until overridden. To set this, type *CONFIGURE Drive 4 (or similar). (CMOS default is 0)

Floppies <n>

This informs ADFS how many floppy drives are connected to the system (minimum 0, maximum 4) in order that references to invalid drives may be trapped and so that ADFS can search these drives for named discs. To set this, type *CONFIGURE Floppies 2 (or similar). (CMOS default is 1)

HardDiscs <n>

This informs ADFS how many Winchester drives are connected to the system (minimum 0, maximum 4). Note that if a Winchester drive is present, then the first Winchester drive is known as drive 4. To set this, type *CONFIGURE HardDiscs 1 (or similar). (CMOS default is 0)

Step <delay> [<drive>]

This informs ADFS what step rates to use for the floppy drives. The delay is given in milliseconds, and the nearest value available on the current hardware will be used. Values supported by the built-in drives are 2, 3, 6 and 12ms. If the drive is given, only that drive's step will be configured, otherwise all of them will.

Entering the ADFS

*ADFS

Syntax: *ADFS

This command selects ADFS as the current filing system. The currently-selected directory, library directory and user root directory all refer to those last used when ADFS was active.

The user root directory

The NetFS has the concept of a user root directory; this is the directory in which the user finds him or herself upon logging on to a fileserver. The ADFS also has the concept of a URD. It may set to any directory on any disc that the filing system knows about.

URDs are useful, as they allow a 'home' directory on the disc in which all of your files (and directories) are kept. This is important if the disc is being shared with other people. If the URD is unset, the ADFS treats references to it (using &) as equivalent to \$. You can set the URD using the *URD command.

Disc specifiers

Many of the commands described below allow discs to be specified. Generally, you can refer to a disc by its physical drive number (eg 0 for the built-in floppy), or by its name. This is set using *NAMEDISC. If the named disc is in a drive, it will be used. Otherwise a `Disc not present` error will be given.

It is possible for machine code programs to trap `Disc not present` errors before they are issued. This allows the user to be prompted to insert the disc into the drive. See the section on upcalls for details.

In fact, disc names may be used in any pathname given to the system. When used in a pathname, the disc name (or number) must be prefixed by a colon. Examples of pathnames with disc specifiers are:


```
*CAT :welcome.fonts
*INFO :4.LIB*.*
```

Note that :drive really means :drive.\$.

A note about changing discs. When you eject a floppy disc from the drive under the ADFS, the system still 'knows' about it. This means that if there are any directories set on that disc (the current directory, user root directory, or library), they will still be associated with it. Thus any attempt to load or run a file will result in a `Disc not present error`.

However, this means that you can replace the disc and still use it, as if it had never been ejected. The same applies to open files on the disc; they remain open and associated with that disc until they are closed.

You can cause the old directories to be overridden by `*MOUNTing` a new disc once it has been inserted. This resets the CSD etc. Alternatively, if you unset the directories (using `*NODIR`, `*NOLIB` and `*NOURD`), then the ADFS will use certain defaults when operations on these are required.

If there is no current directory, the ADFS will use \$ on the default drive. This is the configured default, or the one set by the last `*DRIVE` command.

If there is no library set, then the ADFS will try `&.Library`, `$.Library` and then the current directory, in that order.

If there is no user root directory set, then references to that directory will use \$ on the default drive.

ADFS intrinsic commands

These commands are implemented directly by the ADFS, instead of through FileSwitch. They are only available when the ADFS is the current filing system. For example, if the NetFS is currently selected, you can access ADFS commands using the `-ADFS-` or `ADFS:` prefix:

```
*-adfs-format 0 1
*ADFS:BACK
```

The ADFS has the concept of the default disc, which is used in some commands if an explicit disc reference is omitted. This default is the one on which the current directory (CSD) resides, if one is set, or is the disc in the default drive if not.

***BACK**

Syntax: *BACK

*BACK swaps the previously selected directory (PSD) and the currently selected directory. A common use is for switching between two frequently used directories. You can access the PSD in a pathname, without actually selecting it, by using as the first component.

***BACKUP**

Syntax: *BACKUP <source drive> <destination drive> [Q]

This command backs up the whole contents of a disc onto another one of identical size. If the source drive is the same as the destination, you will be prompted to swap the disc, as necessary.

If you specify the Q option, the application work area is used as a buffer in the backup operation. This allows the backup to be completed with fewer disc accesses, but will corrupt the application program, and control will return to the supervisor (* prompt) when the backup is complete.

***BYE**

Syntax: *BYE

*BYE ends an ADFS session. It ensures that any currently open sequential files are closed, with any data remaining in the associated buffer written to the file(s). In addition, *BYE moves the read/write heads to a 'transit position'. With a Winchester disc unit this operation is vital if the unit is to be moved, otherwise the heads or disc

surface may be damaged. It is advisable to use it at the end of every session to prevent damage should the drive be knocked accidentally.

***COMPACT**

Syntax: *COMPACT [<disc_spec>] [Q]

This command reduces the number of entries in the disc's free space map by moving files around on the disc. The result is that there are fewer and larger areas of contiguous free space. This is important as files have to be stored contiguously on the ADFS.

The drive may be specified as a name, eg `peteDisc` or `:peteDisc`, or as a drive number, eg `0` or `:0`. If the drive is omitted, the default disc is used.

If you specify the Q option, the application workspace will be used during the operation, making it faster. The command will then return to the supervisor prompt.

***DISMOUNT**

Syntax: *DISMOUNT [<disc_spec>]

*DISMOUNT closes all currently open sequential files on either the current or specified drive. In addition, if the current directory, library directory or user root directory is on the dismounted drive, it is unset. Any that are not on the dismounted drive remain intact.

If the dismounted disc is not in a drive, it is forgotten about, so future references to it result in a `Disc not found` error. References to a mounted drive which is not present in a drive cause `Disc not present` errors.

If no drive is specified, the default disc is used. A new disc may be made known to the system using *MOUNT or by setting the current directory by means of *DIR.

***DRIVE**

Syntax: *DRIVE <drive>

*DRIVE sets the drive number that ADFS will use if the currently-selected directory is (or becomes) unset. This is known as the default drive.

***FORMAT**

**Syntax:* *FORMAT <disc_spec> L | D

*FORMAT prepares a floppy disc for use with ADFS. ADFS supports two floppy formats called 'L' and 'D'. The 'L' format is the same as used on the BBC Master Compact ADFS which stores 640 KBytes per floppy, and the 'D' format is a new higher-density format which can store 800 KBytes.

- *Note:* formatting a floppy destroys all data contained on the disc; locking a file is no protection against reformatting! However, you can prevent a floppy from being formatted by moving its write-protect slider to the open position.

***FREE**

Syntax: *FREE [<disc_spec>]

*FREE displays the total amount of free space left and the amount of space used on the given (or default) current disc. The format of the display is:

Bytes free &HHHHHHHH = DDDDDDD

Bytes used &HHHHHHHH = DDDDDDD

Note that the values for the free space are totals and do not take the fragmentation of the free space into account (see *MAP). This means that although there might be, for example, 10240 free bytes on a disc, you can't save a file of that length, because the free space is split into several smaller areas.

*MAP

Syntax: *MAP [<drive>]

*MAP displays a map of the distribution of the total amount of free space, in terms of start sector numbers and lengths as shown below. If the drive is omitted then the current drive is assumed.

```
(   start,   length)
( <aaaaaa>, <11111>) ( <aaaaaa>, <11111>) ...
```

where:

<aaaaaa> is the byte address of a free space
<111111> is the length of the space in bytes

Both numbers are shown in hexadecimal.

The number of entries in the free space map displayed by *MAP is a good guide to the likely need for compaction. The message `Compaction required` is output if the number of entries in the free space map reaches 80 although it is recommended that a disc be compacted if the free space map contains more than 60 entries.

`Compaction required` is also given if you try to save a file which will fit on the disc, but which cannot be saved because no one area of free space is large enough.

*MOUNT

Syntax: *MOUNT [<disc_spec>]

*MOUNT initialises an ADFS disc, ie it reads the free space map and the root directory catalogue into memory and makes the specified drive current. If no disc specification is specified, *MOUNT remounts the current drive.

If the library is unset, it is set to `$.Library`, if this exists. The user root directory is unset.

If a disc is uninitialised, *MOUNT is performed automatically when it is necessary in order to perform another operation.

*NAMEDISC

Syntax: *NAMEDISC <disc_spec> <disc name>

*NAMEDISC renames the disc identified by either the drive number or its current name to the new name given. The disc name can contain between two and ten characters.

You can also spell the command *NAMEDISK.

*NODIR

*NOLIB

*NOURD

These commands allow each of the three main directories to be 'unset'. They perform the opposite functions from the corresponding commands without the 'NO' prefix.

*TITLE

Syntax: *TITLE [<title string>]

*TITLE enables the title string associated with each directory to be changed. By default (ie when it is created), a directory is untitled. The directory title has no significance to the filing system.

<title string> is a sequence of characters which will be written to the title field of the current directory and which will subsequently be displayed for every *CAT command.

The string is terminated by a control character and may contain spaces if required. The title stored in the directory is either truncated or space filled (at the right) to 19 characters.

***URD**

Syntax: *URD [<[wildcarded]directory name>]

If the name is given, *URD sets the user root directory to that name. If no name is given, the user root directory is set to \$ on the current disc. The user root directory may be referenced in pathnames using &.

***VERIFY**

Syntax: *VERIFY [<disc_spec>]

*VERIFY checks that the nominated (or default) disc is readable without error.

ADFS SWI calls

This section lists the SWI calls which are provided by the ADFS module. These calls generally perform very low-level operations, and will not be of interest to the majority of users.

ADFS_DiscOp (&40240) - Perform a miscellaneous disc operation

On entry: R1 = reason code
R2 = disc address
R3 = pointer to memory
R4 = length in bytes

On exit: R0 = 0 if successful, else error pointer
R1 preserved
R2 = address of next byte to be transferred
R3 = pointer to next byte to be transferred
R4 = number of bytes not transferred
V = 1 if there was an error

The reason code is divided into several fields. Bits 0 – 3 give the operation required. Bits 4 – 7 are flags. Bit 8 – 31 are zero if the format of the disc is to be identified by the filing system. Otherwise, they give the word address (ie the actual address DIV 4) of a 64-byte disc record described below.

The disc address in R2 is also divided into fields. Bits 0 – 28 contain the byte address on the disc of the first byte to be accessed. This must be on a sector boundary for commands 0 – 2, and a track boundary for the rest. In addition, it must take into account the defective sector list for Winchester drives. Bits 29 – 31 contain the drive number. This is 0 – 3 for floppy drives and 4 – 7 for Winchesters.

You can calculate the disc address of the first byte given its head/sector/track number from this formula:

$$\text{addr} = ((\text{track} * \text{heads} + \text{head}) * \text{sectorsPerTrack} + \text{sector} - x) * \text{bytesPerSector}$$

Track, head and sector are all counted from zero. X is the adjustment factor for the defective sector list, and is the number of defective sectors before the required sector.

Below is a list of the commands available, given by bits 0 – 3 of R1.

Value	Command	Parameters used
0	Verify	R2, R4
1	Read sectors	R2, R3, R4
2	Write sectors	R2, R3, R4
3	Read track/id	R2, R3
4	Write track	R2, R3
5	Seek	R2
6	Restore	R2
7	Step in (floppy only)	
8	Step out (floppy only)	
15	Specify (hard only)	R2

Command 3 is read track for floppy drives, and read ID on hard discs. Note that only commands 0 – 2 are guaranteed to remain the same for different versions of the hardware. The other calls should be used with caution.

Bit 4 of R1 is usually 0. If it is set, then an alternative defect list for a hard disc transfer is to be used. This list follows the disc record, and may, therefore, only be used if a disc record is specified (bits 8 – 31 are non-zero).

Bit 5 of R1 is usually 0. If it is set, then the meaning of R3 is altered. It does not point to the area of RAM to or from which the disc data is to be transferred. Instead, it points to a word-aligned list of memory address/length pairs. All but the last of these lengths must be a multiple of the sector size. These word-pairs are used for the transfer until the total number of bytes given in R4 has been transferred. On exit, R3 points to the first pair which wasn't fully used, and this pair is updated to reflect the new start address/bytes remaining. This bit may only be set for commands 0 – 2.

If bit 6 is set, then escape conditions are ignored during the operation, otherwise they cause it to be aborted.

If bit 7 is set, then the usual time-out for floppy discs of 1 second is not used. Instead, the ADFS will wait (forever if necessary) for the drive to become ready.

The disc record pointed to by bits 8 – 31 (if non-zero) has the following format:

Byte	Contents
0	Log ₂ of the sector size
1	Number of sectors per track
2	Number of heads (1 for L format, 2 for D format discs)
3	1/2/4 for single/double/quad density
4 – 15	Zeros – these are reserved
16 – 19	Disc size in bytes
20 – 63	Zeros – these are reserved
64...	Alternative defect list, if supplied

A defect list is a list of words. Each word contains the disc address of the first byte of a sector which has a defect. This address is an absolute one, and does not take into

account preceding defective sectors. The list is terminated by a word whose value is &200000XX. The byte XX is a check-byte calculated from the previous words. Assuming this word is initially set to &20000000, it can be correctly updated using this routine:

```

;R2 points to defect list
      MOV   R1, #0                ;Init check
.loop LDR   R2, [R0], #4          ;Get next word
      CMP   R2, #&20000000       ;Last one?
      EORCC R1, R2, R1, ROR #13   ;No, so accumulate check
      BCC   loop                 ;Again
      EOR   R1, R1, R1, LSR #16   ;EOR high and low 16-bit words
      EOR   R1, R1, R1, LSR #8    ;EOR high and low bytes
      AND   R1, R1, #&FF         ;Mask out bits 24-31
      ORR   R2, R2, R1           ;Merge check byte
      STR   R2, [R0, #-4]        ;Save it back

```

Winchester discs contain a &200-byte 'boot block', which contains important information. This occupies sectors &0C and &0D on the disc, and has the following format:

Offset	Contents
&000 - &1AF	Defective sector list
&1B0 - &1BF	Hardware parameters
&1C0 - &1FF	Disc record (see above)

For the HD63463 controller, the hardware parameters have the following contents:

&1B0 – &1B2	Unused
&1B3	Step pulse low
&1B4	Gap 2
&1B5	Gap 3
&1B6	Step pulse high
&1B7	Gap 1
&1B8 – &1B9	Low current cylinder
&1BA – &1BB	Pre-compensation cylinder
&1BC – &1BF	Unadjusted parking disc address

The 'specify disc' command (&0F) sets up the defective sector list, hardware information and disc description from the disc record supplied. Note that in memory, this information would be stored in the order disc record, then defect list/hardware parameters.

ADFS_HDC (&40241) – Set address of hard disc controller etc.

On entry: R0 = address of HDC
R1 = address of poll location for IRQ/DRQ
R2 = bits for IRQ/DRQ
R3 = address to enable IRQ/DRQ
R4 = bits to enable IRQ/DRQ

On exit: –

This call (ADFS 0.04 and above only) sets up the address of the hard disc controller to be used by the ADFS. It can supply an alternative controller to the one normally used on a podule.

ADFS_Drives (&40242) – Read drive configuration

On entry: –

On exit: R0 = current drive
 R1 = *CONFIGURE Floppies value
 R2 = *CONFIGURE HardDiscs value

ADFS_FreeSpace (&40243) – Read free space

On entry: R0 = pointer to disc/drive name

On exit: R0 = total free space on disc
 R1 = largest single area of free space

ADFS error messages

Errors generated by the ADFS have error numbers in the range &90 – &FF. This is given in the bottom byte of the error number (as returned by BASIC's ERR function, for example). The upper three bytes have the value &00010800, this being the code for ADFS errors. You should mask these bytes out when checking against the error numbers given below.

The following errors may be returned by the Advanced Disc Filing System:

Access violation – Error &BD (189)

This error is given when you try to read a file which doesn't have read access, or write a file which doesn't have write access.

ADFS in use – Error &A0 (160)

This error is given when an attempt is made to call the ADFS when that, or another, ADFS routine is already being used. It can occur, for example, if RESET was pressed during an ADFS operation, preventing it from completing properly. It can be cured by performing a hard reset.

ADFS Workspace corrupt – Error &A6 (166)

This error occurs if the ADFS finds that its workspace has been corrupted, for example by a program overwriting the wrong addresses. Perform a hard reset to correct the problem.

Already exists – Error &C4 (196)

This is given when the destination of a *RENAME already exists, and the force option wasn't specified.

Ambiguous disc name – Error &9E (158)

This is given when a wildcarded disc name matches more than one disc that the ADFS currently knows about.

Bad command – Error &FE (254)

This is a FileSwitch error. It is given when an attempt is made to load and run a file, whose name has been given as a * command, and the file does not exist, or is a directory.

Bad disc – Error &9A (154)

This is given when an attempt is made to use a disc which is not in ADFS format.

Bad drive – Error &AC (172)

This is given if you specify a drive number which is too big for the number of configured floppies or Winchester drives, eg :1 on a single floppy system.

Bad free space map – Error &A9 (169)

This is a 'fatal' error indicating either a corruption in RAM (which may normally be cleared by a hard break) or corruption of sectors 0 and 1 on the current disc.

Bad (file) name – Error &CC (204)

This indicates that a filename component longer than ten characters or containing illegal characters was detected in a filename.

Broken directory – Error &A8 (168)

This indicates that the filing system has detected corruption in the format of a directory on the disc. The implication is that either memory has been corrupted or the disc is in an inconsistent state and should be reformatted if possible.

Can't delete current directory – Error &96 (150)

This indicates an attempt to delete the current directory.

Can't delete library – Error &97 (151)

This indicates an attempt to delete the current library directory.

Can't delete user root directory – Error &A2 (162)

This indicates an attempt to delete the current user root directory.

Channel on FileSwitch handle nn – Error &DE (222)

This is a FileSwitch error which is produced when a file handle has been specified which does not correspond to an open file. There is an ADFS version of the error, simply Channel, but this should never occur.

Compaction required – Error &98 (152)

This indicates that the free space on the disc has become too fragmented, ie there are 80 entries in the free space map, or there is not enough contiguous space to perform an operation, but enough total space. The disc must be compacted using *COMPACT.

Directory full – Error &B3 (179)

This message indicates an attempt to create a new entry in a directory which already contains 47 (L) or 77 (D) entries.

Directory not empty – Error &B4 (180)

This message indicates an attempt to delete a directory which still contains objects.

Disc error <nn> at :<d>/<ss> – Error &C7 (199)

This message indicates that the disc controller detected a fault during the last operation:

<nn> is the fault number (in hexadecimal)
<d> is the drive number
<ss> is the sector number <in hexadecimal>

The value <nn> is also given in bits 24 – 29 of the error number.

Disc full – Error &C6 (198)

This message indicates that there is insufficient free space on the disc to allow completion of the current operation.

Disc not found – Error &D4 (212)

This is given when a specified disc name is not known to the system, ie has never been mounted, or has been dismounted since it was used.

Disc not present – Error &D5 (213)

This error is given when an attempt is made to access a disc which is not in any of the drives, although it has been mounted.

Drive empty – Error &D3 (211)

This is given when the ADFS attempts to read a disc (for example to read in a directory), but there is nothing in the drive.

File ‘.’ not found – Error &D6 (214)

This is the FileSwitch version of the Not found error; see below.

File open – Error &C2 (194)

This error occurs if an attempt is made to open, delete or overwrite a file which is already open. *CLOSE or *SHUT may be used to close the file if necessary.

Free space map full – Error &99 (153)

This message indicates that, although there may be free space on the disc, there is no more space available to extend the free space map. The disc must be compacted.

Illegal use of ^ – Error &9C (156)

Self-explanatory.

‘.’ is a directory – Error &A8 (168)

This is given when an attempt is made to perform a byte access operation on a directory. For example, *TYPEing a directory would cause this. It is given by FileSwitch, rather than ADFS.

Locked – Error &C3 (195)

This message indicates an attempt to delete, overwrite or rename a file for which the ‘L’ attribute is set.

Not found – Error &D6 (214)

This error occurs when a file specified, for example as the source file in a *RENAME, is not found. FileSwitch also generates a similar error.

Not same disc – Error &9F (159)

This is given when you try to rename a file across discs. The new and old name must be on the same disc.

Protected disc – Error &C9 (201)

This message indicates an attempt to write to a (floppy) disc which is protected by means of a write protect tab.

Sizes don't match – Error &AD (173)

Same disc – Error &AE (174)

These both refer to errors during a *BACKUP command. Both discs must be of the same format (both L or both D). The second error occurs when, on a single drive backup, you attempt to backup a disc onto itself.

Too many discs – Error &9B (155)

This error is given when an attempt is made to use a new disc when there are already eight discs known to the system which can't be 'forgotten' (because they have current directories or files open on them).

Too many open files – Error &C0 (192)

This error is given when the ADFS cannot allocate enough space to open a file. It is more likely that the FileSwitch limit of 24 files will be exceeded before this happens. FileSwitch gives the same error message and number, except that the top bytes are zero.

Types don't match – Error &C4 (196)

This error occurs when you try to perform an operation on a directory which only applies to files, or vice versa.

Wild cards – Error &FD (253)

This message indicates the illegal use of the wildcard characters in parameters to commands which require explicit references.

THE NETWORK FILING SYSTEM

The Econet is a system which enables several types of computer (including the Archimedes series) to communicate over a low-cost network. File operations are performed on remote 'file server' machines. The NetFS communicates with the file server on the user's behalf. All of the generic filing system commands already described are available to a NetFS user. In addition, the commands listed below are specific to the NetFS.

NetFS intrinsic commands

These commands may only be used when NetFS is the currently-selected filing system, or by prefixing them with `-net-` or `net:` when some other filing system is current.

For example:

```
net#0Z:
net#42.253
```

***BYE**

Syntax: ***BYE** [[:]<file server>]

***BYE** sends a logoff command to the nominated (or currently-selected) file server. You must have already logged on to the nominated file server. Your context (current directory, user root directory and library) on that file server is invalidated. All files

belonging to that file server are flushed to the disc and closed and all handles associated with those files are invalidated.

***FREE**

Syntax: *FREE [<user name>]

*FREE displays your free space currently remaining as well as the total free space for the disc(s) on the currently-selected file server. If an argument is supplied, the free space for the user of that name will be printed out instead.

***FS**

Syntax: *FS [[:]<file server>]

*FS changes your currently-selected file server, restoring your previous context. If no argument is supplied your current file server name and number are printed out, followed by any non-current contexts. This allows you to be logged on to two or more file servers at one time and to change between them. Any open files will be ensured to the current file server before the number is changed. If the argument is a named file server, it must already have been logged onto.

***LOGON**

Syntax: *LOGON [[:]<file server>] <user name> [:] [<password>]

*LOGON sends a logon command to the nominated (or currently-selected) file server, which validates the supplied user name and password against those known. If this succeeds, the file server returns the context appropriate for the user who is logging in. A '?' followed by may be used in the middle of the command in order to hide user name or passwords. See *PASS.

*I AM may be used as an alternative to *LOGON. The former also selects NetFS as the current filing system.

***MOUNT**

Syntax: *MOUNT [[:]<disc name>]

*MOUNT reselects your user root, current directory and library directory on the named (or current) disc.

***PASS**

Syntax: *PASS [:] <old password> <new password>

*PASS allows you to change your password on the currently-selected file server. A ':' followed by may be used in the middle of the command to hide the password. It should be noted that although the ':' may appear anywhere in the line it would be most useful to have it before both the old and new passwords. Whilst the 'invisible' part is being typed U deletes the entire 'invisible' part and Delete deletes the last character.

Naming

As well as being able to supply a filing system name as part of a file name, such as 'Net:&.Fred' it is possible to supply, as part of the filing system name the name or number of a file server, for example 'Net#253:&.Fred' or 'Net#Maths:Program'. This will cause the file to be found (or saved, or whatever) on the given file server. If a name is quoted, that file server must currently be logged on to. If a number is given then the resulting file server must be logged on to, if only part of the number is given then it will be defaulted against the current file server number.

CONFIGURATION COMMANDS***CONFIGURE FS**

This command is used to change the value stored in the configuration memory controlling which file server determines which file is to be the default when the machine is turned on. This value can be either a name of up to sixteen characters or a full network address.

***STATUS FS**

This shows the current stored configuration of the file server; it will be either a number or a name. If it is a name then it will be printed between double quotation marks.

***CONFIGURE LIB**

The state stored in the 'Lib' configuration is used at logon time to decide whether the default library found by the file server is to be used or the alternate library '\$.ArthurLib' is to be searched for and used if it exists. Setting this value to 0 uses the default, setting it to 1 uses '\$.ArthurLib'. The search order for the alternate library is the same as the normal search order for a library, ie the lowest physical disc first.

***STATUS LIB**

The status stored in the 'Lib' configuration is displayed as either <default> or ArthurLib, in single quotation marks.

INTERFACES

There are three ways to interface with the NetFS: through the filing system, through star commands, and by SWI calls. These SWI calls are listed below.

SWI NetFS_ReadFSNumber

R0 <= Station

R1 <= Net

SWI NetFS_SetFSNumber

R0 => Station

R1 => Net

SWI NetFS_ReadFSName

R1 => Buffer
R2 => Size of the buffer
R0 <= Buffer
R1 <= Updated buffer
R2 <= Updated size of the buffer

SWI NetFS_SetFSName

R0 => Buffer address

SWI NetFS_ReadCurrentContext

R0 <= User root directory
R1 <= Currently selected directory
R2 <= Currently selected library

SWI NetFS_SetCurrentContext

R0 => User root directory
R1 => Currently selected directory
R2 => Currently selected library

SWI NetFS_ReadFSTimeouts

R0 <= Transmit count
R1 <= Transmit delay
R2 <= Machine peek count
R3 <= Machine peek delay
R4 <= Receive delay
R5 <= Broadcast delay

SWI NetFS_SetFSTimeouts

R0 => Transmit count
R1 => Transmit delay
R2 => Machine peek count
R3 => Machine peek delay
R4 => Receive delay
R5 => Broadcast delay

SWI NetFS_DoFSOp

R0 => Function
R1 => Buffer address
R2 => Send size
R3 => Receive size
R0 <= Command code
R3 <= Received size

Note that this interface enables interrupts and so cannot be called from within interrupt service code.

Conventions and values

- Station numbers and network numbers are as per Econet.
- Delays are in centi-seconds, and repeat counts are cardinals.
- The DoFSOp SWI is for calling the file server. An example is given below.

```
ReadFileServerVersion
    MOV    r0, #25           ; Command
    ADR    r1, Buffer
    MOV    r2, 0             ; Nothing to send
    MOV    r3, #(?Buffer - 1) ; Lots to receive
    SWI    XNetFS_DoFSOp
    BVS    Error
    MOV    r0, #0           ; Terminate string returned
```

```

        STRB  r0, [ r1, r3 ]      ; One byte past the return size
        MOV   r0, r1
        SWI   XOS_Write0         ; Print it
        BVS   Error

PrintStationNumberOfUser        ; User name pointed to by R0
        ADR   r1, Buffer
        MOV   r2, #0             ; Initial value of index
Loop  LDRB   r3, [ r0 ], #1
        CMP   r3, #" "          ; Check for termination
        MOVL  r3, #13           ; Translate to what the FS wants
        STRB  r3, [ r1, r2 ]    ; Copy into transmit buffer
        ADD   r2, r2, #1        ; Update index, and size to send
        BGT   Loop
        MOV   r0, #24           ; Command
        MOV   r3, #?Buffer
        SWI   XNetFS_DoFSOp
        BVS   Error
        LDRB  r3, [ r1, #1 ]    ; Pickup station number
        LDRB  r4, [ r1, #2 ]    ; Pickup network number
        STMFD r13!, { r3, r4 } ; Deposit in stack frame
        MOV   r0, r13          ; Pointer to value for conversion
        MOV   r2, #?Buffer     ; Destination size
        SWI   XOS_ConvertNetStation
        ADD   r13, r13, #8      ; Dispose stack frame
        SWIVC XOS_Write0       ; Display output
        SWIVC XOS_NewLine
        BVS   Error

```

Implementation limits

There is no machine peek prior to logon so these timeouts are ignored.

There is no mechanism for manipulating context, so the SWIs
 NetFS_ReadCurrentContext and NetFS_SetCurrentContext are not implemented.

NETPRINT, THE NETWORK PRINTING SYSTEM

Naming

The network printing system is actually a filing system, and as such it can be used by giving it's name as part of a file name, for example:

```
*save Netprint:fred
```

although with current implementations the file name is ignored and the 'NetPrint:' part is used to send the data to the network printer. As well as save operations the NetPrint filing system is also capable of opening files and taking data. This means that the operating system can spool to NetPrint:.

Whenever a file is opened or saved onto NetPrint: the current printer server is used. This printer server can be set using a star command, as well as be defaulted from a stored configuration. This default can also be overridden by supplying the printer server number, or name, as part of the file name. For example:

```
NetPrint#234:.
```

This example would then send the print to the printer server at station 234. As usual a full network number can be specified. For example:

```
Netprint#2.235:
```

Also since printer servers can be named, you can supply the printer name, rather than the number. For example:

```
NetPrint#Epson:
```

```
NetPrint#Daisy:
```

The NetPrint filing system supports the OS_File operation Save and the OS_Find operation OpenOut as well as OS_BPut and OS_GBPB writes (but not backwards).

Star commands

*PS

The PS command takes a single argument, either a name or a network address, and this is then the current printer server number. If a name is given then the name is 'looked up' immediately and the current printer server is set to the number of the first named printer of that name to reply. If the lookup fails the number will not change.

*SetPS

The PS command takes a single argument, either a name or a network address, and this is then the current printer server name or number. If a name is given then the name is stored as the current printer server. Only when the file is actually saved or opened is the name 'looked up' and bound to a number.

Configuration demands

*Configure PS

This is used to change the value stored in the configuration memory, ie printer is to be the default when the machine is turned on. This value can be either a name of up to six characters or a full network address.

*Status PS

This shows the current stored configuration of the printer server. It will be either a number or a name. If it is a name then it will be printed between double quotation marks.

Linking NetPrint to *FX 5 4 and VDU 2

There are system variables that connect the VDU print streams to files; an example of this is the default value set up by NetPrint upon it's initialisation. This is PrinterType\$4 and it's value is NetPrint:. This value could be changed to indicate a particular printer:

NetPrint#Epson:

and another variable set up to contain a different value:

PrinterType\$3 = NetPrint#2.235

so that swapping between printers is done with an FX call eg *FX 5 4 or *FX 5 3.

INTERFACES

There are three ways to interface with the NetPrint: through the filing system, star commands or by SWI calls. These SWI calls are listed below.

SWI NetPrint_ReadPSNumber

R0 <= station

R1 <= net

SWI NetPrint_SetPSNumber

R0 => station

R1 => net

SWI NetPrint_ReadPSName

R1 => buffer

R2 => size of buffer

R0 <= buffer

R1 <= updated buffer

R2 <= updated size of buffer

SWI NetPrint_SetPSName

R0 => buffer address

SWI NetPrint_ReadPSTimeouts

R0 <= transmit count
 R1 <= transmit delay
 R2 <= machine peek count
 R3 <= machine peek delay
 R4 <= receive delay
 R5 <= broadcast delay

SWI NetPrint_SetPSTimeouts

R0 => transmit count
 R1 => transmit delay
 R2 => machine peek count
 R3 => machine peek delay
 R4 => receive delay
 R5 => broadcast delay

Conventions and values

Station numbers and network numbers are as per Econet.

Delays are in centi-seconds, and repeat counts are cardinals.

Implementation limits

There is no machine peek prior to connect, so these timeouts are ignored.

WRITING YOUR OWN FILING SYSTEM

Relocatable modules, which are described in more detail later in this manual, can be used to provide operating system extensions, including filing systems. Both the Advanced Disc Filing System and Net Filing System are implemented as modules. This section describes the low level calls which are used by the filing system manager (FileSwitch) and which have to be implemented by anyone wishing to write his or her own filing system. Before using this information, you need to be familiar with the assembler and use of modules which are described in the following chapters.

Declaring, selecting and removing an FS

OS_FSCControl &29 (41) – Filing system control

This SWI performs the fundamental operations of declaring a module to be a filing system and selecting or deselecting it. The reason codes to do these are as follows:

R0 = &0C (12) Add filing system

On entry: R1 = pointer to module base address
R2 = offset of the filing system information block
R3 = private word pointer

On exit: V set on error

This should be called from the initialisation code of your relocatable module to inform the operating system that your module contains a filing system.

The filing system information block, pointed to by R1+R2 contains the following:

Offset Contains

&00	Offset of filing system name (null terminated)	
&04	Offset of filing system startup text (null terminated)	
&08	Offset of routine to open files	(FSEntry_Open)
&0C	Offset of routine to get bytes from media	(FSEntry_GetBytes)
&10	Offset of routine to put bytes to media	(FSEntry_PutBytes)
&14	Offset of routine to control open files	(FSEntry_Args)
&18	Offset of routine to close open files	FSEntry_Close)

&1C	Offset of routine to do file operations	(FSEntry_File)
&20	Word of filing system type information	
&24	Offset of routine to do various FS operations	(FSEntry_Func)
&28	Offset of routine to do multi-byte operations	(FSEntry_GBPB)

The GBP entry (&28) is only required if the filing system supports non buffered I/O (eg the vdu: device uses this).

The information word (&20) tells the filing system manager various things about the filing system:

Bit	Meaning if set
31	Special fields are supported
30	Streams are interactive
29	Filing system supports null length filenames

Special fields start with *. They start at the end of the filing system name and end with the character which marks the end of the name. For example, a NetFS pathname might be:

```
-net#1.254-disc1.fred
```

or

```
net#40.::disc1.jim
```

The special field here is 1.254, and gives the network and station number of the fileserver to use. Special fields are passed to the filing system as null-terminated strings, with the '#' and '.' stripped off. If no special field is specified in a pathname, the appropriate register in the FS routine is set to zero. See below for details of which calls may take special fields.

In addition, bits 0 – 7 contain the filing system identification number. Currently used ones are:

File system Number

NetFS	5
ADFS	8
VFS	10
NetPrint	12
Null	13
Printer	14
Serial	15
Vdu	17
RawVdu	18
Kbd	19
RawKbd	20
DeskFS	21

Acorn FS = 56
Spontaneous FS = 66

Rain = 23
Idle FS = 49

For an allocation, contact Acorn Computers in writing.

The private word pointer passed to the SWI in R3 is the value which will be passed in R12 on entry to any of your filing system entry points or filing system star command code entries.

R0 = &0E (14) *Filing system selection*

On entry: R1 = pointer to filing system name, or FS number

On exit: V set on error

If R1 is < &100 it is taken to be a number. Otherwise, it is assumed to be a pointer to a control character-terminated name. To be able to select a filing system as the current filing system, one of your module's star command handlers must use this call in response to a suitable utility star command, such as *ADFS or *NET.

Possible errors are Filing system 'xxx' not present and Bad filing system name.

R0 = \$10 (16) Filing system removal

On entry: R1 = pointer to filing system name

On exit: -

This call removes the filing system from the list held by the filing system manager. You may wish to use it in the finalise entry of your module.

Modules should not complain about errors in filing system removal. Otherwise, it will not be possible to reinitialise the module after reinitialising the filing system manager.

Filing system interface calling conventions

Routines called by the filing system manager are always entered in supervisor mode. This enables them to access hardware devices directly and to set up FIQ registers as necessary.

R13 in supervisor mode is used as the system stack. This may be used by the filing system. It is only guaranteed to be 1024 bytes deep. When the filing system is entered, take care not to push too much onto it. The stack base is on a 1Mbyte boundary. Hence, to determine how much stack space there is left for your use, use the following code:

```
MOV   R0, R13, LSR #20           ;Get Mbyte value of SP
SUB   R0, R13, R0, LSL #20       ;Sub it from actual value
```

You may move the stack pointer downwards by a given amount and use that amount of memory as temporary workspace. However, interrupt processes are allowed to use the supervisor stack so you must leave enough room for these to operate. Similarly, if you call any operating system routines, you must give them enough stack space.

R12 on entry to the filing system is set to the value of R3 passed to the filing system manager in the SWI OS_FSCControl call to initialise the filing system. Conventionally, this is used as a pointer to your private word. In this case, module entries should contain the following:

LDR R12, [R12]

to load the actual address into the register.

Filing system routines do not need to preserve any registers other than R13.

If a routine wishes to return an error, it should return to the filing system manager with V set and R0 pointing to a standard format error block.

R0 = 17 Add secondary module

On entry: R1 = filing system name
R2 = secondary system name
R3 = secondary module workspace pointer

This call enables you to locate and search another module.

The filing system interface calls

FSEntry_Open

On entry: R0 = what file is being opened for
R1 = pointer to filename (null terminated)
R3 = handle to this file
R6 = pointer to special field if present, otherwise 0

On exit: R0 = information word
R1 = handle returned by filing system (0 if not found)
R2 = buffer size to use (0 if file unbuffered)
R3 = file extent (buffered files only)
R4 = space currently allocated to file (buffered files)

This call is used to open a file for read or write, and can create it if necessary. The value given in R0 has the following meaning:

Value	Meaning
0	Open for read
1	Create and open for update
2	Open for update

The pointer to the filename is passed in R1. If the file is being created and opened for update (R1=1), then this filename may not be wildcarded.

On entry, R3 contains the FileSwitch handle for the file, which is a small integer (typically in the range 1 – 24). The filing system may want to make a note of it when the file is opened, in case it needs to refer to files by their FileSwitch handles (eg files are automatically closed on a *DISMOUNT). It is the FileSwitch handle that the user sees.

The filing system manager treats any 32-bit handle (except 0 and -1) which is returned by the filing system as being valid. It uses this handle for any further calls to the filing system regarding this particular file. A handle of zero means that no file is open; a handle of -1 is used to indicate 'unset' directory contexts (see FSEntry_Func).

If the object is a directory, it may only be opened for reading. However, bytes will not be requested from it. The use of this is for compatibility with existing programs which use this as a method of testing the existence of an object. This is also used to open new directory contexts which may be written via FSEntry_Func.

The information word returned in R0 contains the following bits:

Bit	Meaning if set
31	Write permitted to this file
30	Read permitted from this file
29	Object is a directory
28	Unbuffered OS_GBPB supported (streams-type devices only)
27	Stream is interactive

FSEntry_GetBytes

This call is used to get a single byte or a group of bytes from an open file. There are two distinct cases to consider, depending on whether the file was opened as buffered or unbuffered:

Get bytes from a buffered file

On entry: R1 = file handle
R2 = memory address to put data
R3 = number of bytes to read
R4 = file offset to get data from

On exit: -

This call reads a number of bytes (which must be a multiple of the buffer size for this file) and places them in memory. The file offset from which to read data should also start at a multiple of the buffer size for this file.

Get byte from an unbuffered file

On entry: R1 = file handle

On exit: R0 = byte read, C clear
R0 = undefined, C set if attempting to read at end of file

This call is used to get a single byte from an unbuffered file from the position given by the file's sequential pointer. The pointer must be incremented by one, unless the end of the file has been reached.

FSEntry_PutBytes

This call is used to put a single byte or group of bytes to a file. Again, there are two distinct cases to consider:

Put bytes to a buffered file

On entry: R1 = file handle
 R2 = memory address to take data from
 R3 = number of bytes to put to file
 R4 = file offset to put data to

On exit: -

This call is used to take a number of bytes, which will be a multiple of the buffer size for this file, and place them in the file at the specified file offset. The offset at which to put the data will also start at a multiple of the buffer size for this file.

Put byte to an unbuffered file

On entry: R0 = byte to put to file (top 24 bits zero)
 R1 = file handle

On exit: -

This call is used to put a single byte to an unbuffered file at the position given by the file's sequential file pointer. The sequential file pointer must be advanced by one.

FSEntry_Args

Various calls are made through this entry point to deal with controlling open files. The actions are specified by R0 as follows:

R0 = 0 Read sequential file pointer

On entry: R1 = file handle

On exit: R2 = sequential file pointer

This call is used to read the sequential file pointer for a given file. Only filing systems which use unbuffered files should support this call.

R0 = 1 Write sequential file pointer

On entry: R1 = file handle
R2 = new sequential file pointer

On exit: -

This call is used to alter the sequential file pointer for a given file. The file must be extended with zeros if the new pointer is greater than the current file extent. Only filing systems which use unbuffered files should support this call.

R0 = 2 Read file extent

On entry: R1 = file handle

On exit: R2 = file extent

This call is used to read the extent of a given file. Only filing systems which use unbuffered files should support this call.

R0 = 3 Write file extent

On entry: R1 = file handle
R2 = new file extent

On exit: -

For buffered files, this call is only issued internally by the filing system manager in order to set the real file extent just prior to closing an open file.

For unbuffered files, this call is passed directly through from the user. The file must be extended with zeros if the new extent is greater than the current file extent.

R0 = 4 Read size allocated to file

On entry: R1 = file handle

On exit: R2 = size allocated to file by filing system

This call is used to read the size allocated to a given file. All filing systems must support this call.

R0 = 5 *EOF check*

On entry: R1 = file handle

On exit: R2 <> 0 if file pointer is at end of file, 0 otherwise

This call is used to determine whether the sequential pointer for a given file is at the end of the file or not. Only filing systems which use unbuffered files should support this call.

R0 = 6 *Flush file buffer*

On entry: R1 = file handle

On exit: -

This call is used to flush any buffered data for a file. Only filing systems which use unbuffered files should support this call.

R0 = 7 *Ensure file size*

On entry: R1 = file handle
R2 = size of file to ensure

On exit: R2 = size of file actually ensured

This call is used to ensure that a file is of at least the given size. If the file is extended by this call, no data need be transferred unless the file is unbuffered or ignores the write zeros call, in which case the file must be zero-extended. All filing systems must support this call.

R0 = 8 Write zeros to file

On entry: R1 = file handle
R2 = file address to write zeros at
R3 = number of zero bytes to write

This call is used to add a number of zeros to a file. It is only issued by the filing system manager when extending a file. Filing systems may ignore this call if they make sure that ensuring a file size adds zeros to the end. Only filing systems which use buffered files should support this call.

R0 = 9 Read file datestamp

On entry: R1 = file handle

On exit: R2 = load address of file (or 0)
R3 = execution address of file (or 0)

This call is used to read the date/time stamp for a given file. The bottom four bytes of the date/time stamp are stored in the execution address of the file. The most significant byte is stored in the least significant byte of the load address. All filing systems must support this call. If a filing system cannot stamp an open file given its handle, then it should return R2 and R3 set to zero.

FSEntry_Close

On entry: R1 = file handle
R2 = new load address to associate with file
R3 = new execution address to associate with file

On exit: -

This call is used to close an open file and put a new date/time stamp on it. If the load address and execution address are zero then the file should not be restamped.

Note that *CLOSE (ie close all open files) is performed by the filing system manager which passes the handles, one at a time, to the filing system for closing. Filing systems should not try to support this themselves.

FSEntry_File

This call is used to perform operations on whole files depending on the value of R0 as follows:

R0 = *0FF* Load file

On entry: R1 = pointer to wildcarded filename (null terminated)
 R2 = address to load file
 R3 = instructions for loading
 R6 = pointer to special file if present; otherwise zero

On exit: R0 is corrupted
 R2 = load address
 R3 = execution address
 R4 = file length
 R5 = file attributes
 R6 = pointer to a filename for printing *OPT 1 info

The filename pointed to by R1 must be the name of an existing file. If the file does not exist or is a directory then an error should be given.

The address at which to load the file is passed in R2. Whether or not this is used depends on R3 as follows:

Value	Meaning
0	Load file at address given by R2
Not 0	Load file at its own address

The filename pointed to by R6 on exit should be the non-wildcarded 'leaf' name of the file. That is, if the filename given on entry was \$. !b*, and the file accessed was the boot file, R6 should point to the filename !BOOT.

R0 = 0 Save file

On entry: R1 = pointer to filename (null terminated)
R2 = load address to associate with file
R3 = execution address to associate with file
R4 = start address in memory of data
R5 = end address in memory plus one
R6 = pointer to special field if present, otherwise 0

On exit: R6 = pointer to a filename for printing *OPT 1 info

This call saves an area of memory to a file. An error such as `File Locked` should be returned if the specified file could not be saved.

R0 = 1 Write catalogue information

On entry: R1 = pointer to wildcarded filename (null terminated)
R2 = new load address to associate with file
R3 = new execution address to associate with file
R5 = new attributes for file
R6 = pointer to special file if present, otherwise 0

On exit: -

This call updates the catalogue information. An error occurs if the object is a directory but not if the object does not exist.

R0 = 2 Write load address

On entry: R1 = pointer to wildcarded filename (null terminated)
R2 = new load address to associate with file
R6 = pointer to special file if present, otherwise 0

On exit: -

This call alters the load address for a file. An error occurs if the object is a directory, but not if the object does not exist.

R0 = 3 *Write execution address*

On entry: R1 = pointer to wildcarded filename (null terminated)
 R3 = execution address to associate with file
 R6 = pointer to special field if present, otherwise 0

On exit: -

This call alters the execution address for a file. An error occurs if the object is a directory, but not if the object does not exist.

R0 = 4 *Write attributes*

On entry: R1 = pointer to wildcarded filename (null terminated)
 R5 = new attributes to associate with file
 R6 = pointer to special field if present, otherwise 0

On exit: R6 = pointer to a filename for printing *OPT 1 info

This call alters the attributes of a file. An error occurs if the object is a directory, but not if the object does not exist.

R0 = 5 *Read catalogue information*

On entry: R1 = pointer to filename (null terminated)
 R6 = pointer to special field if present, otherwise 0

On exit: R0 = object type
 R2 = load address
 R3 = execution address
 R4 = file length
 R5 = file attributes

This call returns the catalogue information for a file. An error does not occur if the object does not exist.

R0 = 6 *Delete object*

On entry: R1 = pointer to filename (null terminated)
R6 = pointer to special field if present, otherwise 0

On exit: R0 = object type
R2 = load address
R3 = execution address
R4 = file length
R5 = file attributes

This call deletes an object. An error occurs if the object is locked against deletion, but not if the object does not exist. The results refer to the object that was deleted.

R0 = 7 *Create file*

On entry: R1 = pointer to filename (null terminated)
R2 = load address to associate with file
R3 = execution address to associate with file
R4 = start address in memory of data
R5 = end address in memory plus one
R6 = pointer to special field if present, otherwise 0

On exit: R6 = pointer to a filename for printing *OPT 1 info

This call creates a file with a given name. R4 and R5 are used only to calculate the length of the the file to be created. If the file currently exists and is not locked, the old file is first discarded. An error occurs if the file could not be created.

R0 = 8 *Create directory*

On entry: R1 = pointer to directory name (null terminated)
R4 = number of entries (0 for default)
R6 = pointer to special field if present, otherwise 0

On exit: -

This call creates a directory. If the directory already exists then it is kept intact and nothing happens. An error occurs if the file could not be created.

FSEntry_Func

Various calls are made through this entry point to deal with assorted filing system control. The actions are specified by R0 as given below. Note that 'dirname' means 'directory name'.

R0 = 0 Set current directory

On entry: R1 = pointer to wildcarded dirname (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit: -

This call is used to set the current directory to the one identified by the dirname given. If the dirname is null, the directory should be set to the filing system default (typically the same as the user root directory).

R0 = 1 Set library directory

On entry: R1 = pointer to wildcarded dirname (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit: -

This call is used to set the current library directory to the one identified by the dirname given. If the dirname is null, the library directory is set to the filing system default.

R0 = 2 Catalogue directory

On entry: R1 = pointer to wildcarded dirname (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit: -

This call is used to catalogue the directory identified by the dirname given. If the dirname is null, the current directory should be catalogued. (This corresponds to the *CAT command.)

R0 = 3 Examine current directory

On entry: R1 = pointer to wildcarded dirname (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit: -

This call is used to print information on all the objects in the directory identified by the dirname given. If the dirname is null, the current directory should be examined. (This corresponds to the *EX command.)

R0 = 4 Catalogue library directory

On entry: R1 = pointer to wildcarded dirname (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit: -

This call is used to catalogue the specified subdirectory relative to the current library directory. If the dirname is null, the current library directory should be catalogued. (This corresponds to the *LCAT command.)

R0 = 5 Examine library directory

On entry: R1 = pointer to wildcarded dirname (null terminated)
R6 = pointer to special field if present, otherwise zero

On exit: -

This call is used to print information on all the objects in the specified subdirectory relative to the current library directory. If the dirname is null, the current library directory should be examined. (This corresponds to the *LEX command.)

R0 = 6 Examine object(s)

On entry: R1 = pointer to wildcarded pathname (null terminated)
R6 = pointer to special field if present, otherwise zero.

On exit: -

This call is used to print information on all the objects matching the wildcarded pathname given, in the same format as for Examine directory. (This corresponds to the *INFO command.)

R0 = 7 Set filing system options

On entry: R1 = option (or 0)
R6 = parameter

On exit: -

This call is used to set filing system options. An option of 0 means reset all filing system options to their default values. (This corresponds to the *OPT command.)

R0 = 8 Rename object

On entry: R1 = pointer to first pathname (null terminated)
R2 = pointer to second pathname (null terminated)
R6 = pointer to first special field if present, otherwise 0
R7 = pointer to second special field if present, else 0

On exit: R1 = 0 if rename performed (<>0 otherwise)

This call is used to attempt to rename an object. If the rename is not 'simple', (ie just changing the file's catalogue entry) R1 should be returned with a value other than zero. In this case, the filing system manager copy/delete will be used to do the rename.

R0 = 9 Access object(s)

On entry: R1 = pointer to wildcarded pathname (null terminated)
R2 = pointer to access string (null terminated)

On exit: –

This call is used to give the requested access to all objects matching the wildcarded name given. (This corresponds to the *ACCESS command.)

R0 = 10 Boot filing system

On entry: –

On exit: –

The filing system should perform its boot action on this call. For example, ADFS examines the boot option (as set by *OPT 4) of the disc in the configured drive and acts accordingly, for example *RUN !BOOT if boot option 2 is set; whereas NetFS attempts to logon as the boot user to the configured file server.

*R0 = 11 Read name and boot (*OPT 4) option of disc*

On entry: R2 = memory address to put data

On exit: –

This call is used to obtain the name of the disc and its boot option. This data should be returned in the area of memory pointed to by R2, in the following format:

<name length byte><disc name><boot option byte>

R0 = 12 Read current directory name and privilege byte

On entry: R2 = memory address to put data

On exit: –

This call is used to obtain the name of the currently-selected directory, and privilege status in relation to that directory. This data should be returned in the area of memory pointed to by R2, in the following format:

<zero byte><name length byte><current directory name><privilege byte>

The privilege byte is &00 if you have 'owner' status (ie you can create and delete objects in the directory) or &FF if you have 'public' status (ie are prevented from creating and deleting objects in the directory). On ADFS, you always have owner status.

R0 = 13 *Read library directory name and privilege byte*

On entry: R2 = memory address to put data

On exit: -

This call is used to obtain the name of the library directory, and privilege status in relation to that directory. This data should be returned in the area of memory pointed to by R2, in the following format:

<zero byte><name length byte><library directory name><privilege byte>

R0 = 14 *Read directory entries*

On entry: R1 = pointer to wildcarded dirname (null terminated)
 R2 = memory address to put data
 R3 = number of object names to read
 R4 = offset of first item to read in directory
 R5 = buffer length
 R6 = pointer to special field if present, otherwise zero

On exit: R3 = number of names read
 R4 = offset of next item to read in directory (-1 if end)

This call is used to read the names of entries in a directory into an area of memory pointed to by R2. If the directory name is null, then the currently-selected directory

should be read. The names are returned in the buffer as a list of null terminated strings.

RO = 15 Read directory entries and information

On entry: R1 = pointer to wildcarded dirname (null terminated)
R2 = memory address to put data
R3 = number of object names to read
R4 = offset of first item to read in directory
R5 = buffer length
R6 = pointer to special field if present, otherwise zero

On exit: R3 = number of records read
R4 = offset of next item to read in directory (-1 if end)

This call is used to read the names of entries (and their file information) in the given directory into an area of memory pointed to by R2. If the directory name is null, then the currently-selected directory should be read. The names and information are returned in records, with the following format:

Offset	Contents
&00	Load address
&04	Execution address
&08	Length
&0C	Attributes
&10	Object type
&14	Object name (null terminated)

Each record is word-aligned.

RO = 16 Shut down

On entry: -

On exit: -

On this call, the filing system should attempt to go into as dormant a state as possible. For example, it should place Winchester drives in their transit positions, etc. All files will have been closed by the filing system manager before this call is issued.

R0 = 17 Print start up banner

On entry: -

On exit: -

This call is used when the filing system is started, and the filing system start up text offset value (in the filing system information block) is -1. This is to allow filing systems to print a message that may vary, such as *Acorn Econet* or *Acorn Econet no clock*.

R0 = 18 Set directory contexts

On entry: R1 = new currently-selected directory handle (0 = no change, -1 for 'Unset')
 R2 = new user root directory handle (0 = no change, -1 for 'Unset')
 R3 = new library handle (0 = no change, -1 for 'Unset')

On exit: R1 = old selected directory handle (-1 if 'unset')
 R2 = old user root directory handle (-1 if 'unset')
 R3 = old library handle (-1 if 'unset')

This call is used to redefine (or read) the currently-selected directory, user root directory and library handles.

R0 = 19 Read directory entries and information

On entry: R1 = pointer to wildcarded dirname (null terminated)
 R2 = memory address to put data
 R3 = number of object names to read
 R4 = offset of first item to read in directory
 R5 = buffer length
 R6 = pointer to special field if present, otherwise zero

On exit: R3 = number of records read
R4 = offset of next item to read in directory (-1 if end)

This call reads the names of entries (and their file information) in the given directory into an area of memory pointed to by R2. If the directory name is null, then the currently-selected directory should be read. The names and information are returned in records, with the following format:

Offset	Contents
&00	Load address
&04	Execution address
&08	Length
&0C	Attributes
&10	Object type
&14	S.I.N
&18	Data in 5-byte format
&1D	Object name (null terminated)

Each record is word-aligned.

This chapter describes the Archimedes' memory usage. In many environments, such as BASIC and C, you can use the language's intrinsic memory allocation routines, and you won't have to worry about where your next byte is coming from. Similarly, small, transiently loaded utilities may not require any memory over the 1024 bytes they are automatically allocated.

Other programs, though, such as filing systems, specialised VDU drivers (such as the font manager), etc. will require arbitrary amounts of memory, which can be freed after use. The memory manager provides simple allocation and deallocation facilities. Relocatable modules can use this manager either directly or indirectly using the `OS_Module` call.

In addition to the 'heap' RAM allocated by the memory manager, there are other, less important memory resources. One of these is the battery-backed CMOS RAM used to hold default system parameters. The user interface to this RAM is provided by the `*CONFIGURE` and `*STATUS` commands. Modules, applications and users may use spare locations in CMOS RAM for their own purposes. Application and module writers should request allocations in writing from Acorn Computers.

The screen RAM is also available as a temporary buffer. There are quite severe restrictions on when the RAM may be used: see the call `OS_ClaimScreenMemory` for details.

Logical RAM

This chapter is concerned only with the bottom 32 Mbytes of the memory map since this is the area which contains the logically mapped RAM.

The physical memory of the machine cannot be accessed directly in user mode. Instead it is mapped onto pages of the 'logically addressed' RAM and it is these logical pages which the user accesses. The physical RAM is always divided into 128 pages. As the amount of RAM in the machine varies, the size of the page or the numbers of pages used changes as shown below.

300 Series

RAM (Mbytes)	Pages used	Size of page (Kbytes)
0.5	64	8
1	128	8

400 Series

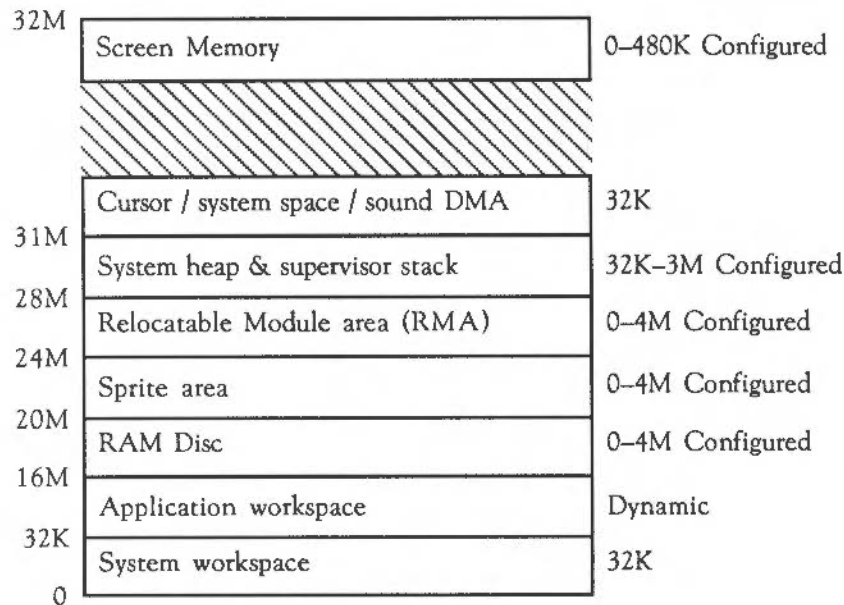
RAM (Mbytes)	Pages used	Size of page (Kbytes)
1	32	32
2	64	32
4	128	32

The MEMC contains a 'logical to physical address translator'. This has a table of 128 entries, one for each page of physical RAM, showing where the page is mapped to in logical RAM. Whenever a logical page is accessed, this translator attempts to convert the logical page number into a physical page number by looking for the address given in each of the table entries.

Since there are 32 Mbytes of logical RAM and a maximum of 4 Mbytes of physical RAM, the area of logically mapped RAM obviously contains far more than 128 pages, so it is possible to attempt to access memory which is not mapped to physical RAM. If this happens, an error message is displayed, such as `Abort on data transfer at <addr>`.

Logical memory map

The organisation of the logical address space is as follows:



The memory map is set up on hard reset as follows:

- The permanent 32K allocations for system workspace at addresses &0000000 and 31M are made.
- Then the configured amounts of space for the various adjustable size regions are allocated: the screen, the system heap, the RMA, etc. Note that some of these have minimum values that will be used regardless of the configured value. For example, the RMA will always have enough pages to support all the built-in ROM modules.
- When all of the dynamic allocation is complete, the rest is allocated to the application workspace, from address &8000 up. While no application is running (ie in the supervisor prompt), the memory map can be altered as required. For example, if you load a module from disc and the RMA isn't big enough to hold it, the size of the RMA will be increased by an appropriate amount. The OS can only do this when there is no application active, as the extra memory has to be

taken from the application workspace. Most programs don't react too kindly to large areas of their memory allocation disappearing.

Here is an example of how memory might be allocated given some typical RAM size allocations on an A310 (8K page size):

Area	Pages	Page size	Total
FontSize	20	4K	0
RamFsSize	0	8K	0
RMASize	16	8K	128K
ScreenSize	20	8K	160K
SpriteSize	10	8K	80K
SystemSize	4	8K	32K+32K
System workspace			32K
Cursor etc. workspace			32K
Total			496K
Application area			1024K - 496K = 528K

- Note: the FontSize doesn't add anything to the total as the font manager takes the required space from the RMA. If the configured RMA size isn't enough to claim the font space, the RMA will be made bigger, as no application is running when the font manager makes its claim.

A configured screen size of 0 means 'default for this machine', which is 160K on an A310 (see *CONFIGURE ScreenSize). The size of the system area (at 28M) is always at least 32K. 8K of this is used for the system stack. The rest is for OS variable storage (eg alias variables) and module information. The configured amount is added to the 32K initially allocated.

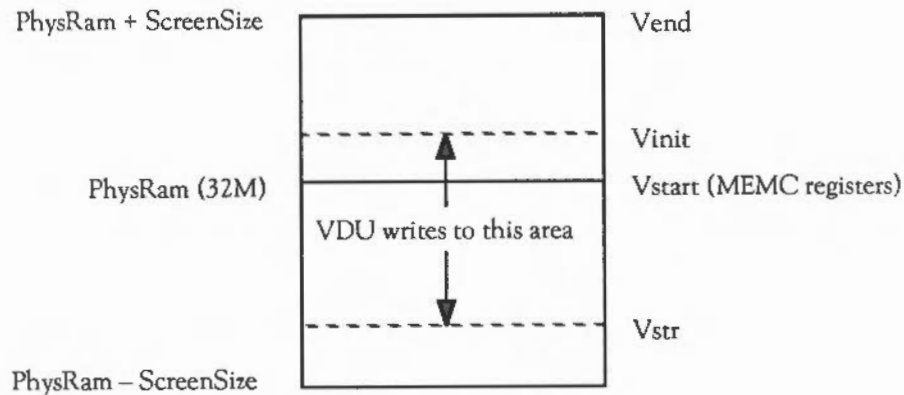
Memory protection

You have read/write access to the whole of the logically mapped RAM. However, only the application workspace and RMA should be accessed directly. It is very dangerous to write to any other areas, or rely on certain locations containing given

information, as these are subject to change. You should always use OS routines to access operating system workspace.

Screen memory

Hardware scrolling is implemented by having the screen workspace at the end of logical memory, adjacent to the corresponding physical RAM banks which are mapped onto those addresses. This means that there are two adjacent copies of the screen memory as follows:



The screen can, therefore, be scrolled vertically by altering the VDU driver screen start address as shown above. This is usually performed automatically and you don't have to concern yourself with it.

The screen-size is configurable in units of one page (8K or 32K). Hence for a 20K screen on a 400 series machine, 32K will have to be used since it is the next highest multiple of 32K. For an 80K screen, 96K should be used, etc. In addition, if you want to use multiple banks of screen memory (eg for animation), enough memory must be reserved for each bank.

Because the total screen memory is often much less than what is required at a given time, a facility is available whereby the 'extra' RAM can be claimed for short

periods. It can be used as a buffer, in a data transfer operation, for example. The SWI documented below is used to claim and release the screen memory.

OS_ClaimScreenMemory &41 (65)

- On entry:* R0 = 0 for release, 1 for claim
R1 = length required
- On exit:* C=0 means memory was claimed successfully
R1 = length available
R2 = start address
C=1 means memory could not be claimed
R1 = length that is available

There are several restrictions to the use of screen memory. It can only be claimed by one 'client' at a time, who gets all of it. It can only be claimed if no bank other than bank 1 has been used. You can't claim it, for example, if the shadow bank has been used.

While you have claimed the screen memory, you must not perform any action which might causes the screen to scroll. This means avoiding the use of routines which might cause screen output.

It is important to release the memory after it has been used.

NON-VOLATILE MEMORY (CMOS RAM)

240 bytes of non-volatile memory are provided. Some of these are reserved since they hold default values for certain parameters and are set using *CONFIGURE. The full list is given below:

Location	Function
0	Econet station number
1	Econet file server station ID (0 => name configured)
2	Econet file server net number (or first char of name)
3	Econet printer server station id (0 => name configured)
4	Econet printer server net number (or first char of name)
5	Default filing system number
6 - 9	Reserved for Acorn use
10	Screen info: <ul style="list-style-type: none"> Bits 0 - 3 screen mode number. This is held in 5 bits. The fifth bit is bit 133 Bit 4 TV interlace (first *TV parameter) Bits 5 - 7 TV vertical adjust (signed three-bit number)
11	Shift, Caps mode: <ul style="list-style-type: none"> Bits 0 - 2 reserved Bits 3 - 5 ShCaps (001), NoCaps (010), Caps (100) Bit 6 - 7 reserved
12	Keyboard auto-repeat delay
13	Keyboard auto-repeat rate
14	Printer ignore character
15	Printer information: <ul style="list-style-type: none"> Bit 0 reserved Bit 1 0 => Ignore, 1 => NoIgnore Bits 2 - 4 serial baud rate (0=75, ..., 7=19200) Bits 5 - 7 printer type
16	Miscellaneous flags <ul style="list-style-type: none"> Bit 0 reserved Bit 1 0 => Quiet, 1 => Loud Bit 2 reserved Bit 3 0 => Scroll, 1 => NoScroll Bit 4 0 => NoBoot, 1 => Boot Bits 5 - 7 serial data format (0..7)
17 - 29	Reserved for Acorn use
30 - 45	Reserved for the user
46 - 111	Reserved for applications
112 - 127	Reserved for operating system software

Location	Function
128	Current year
130	Last month
131	'We've-had-the-29th-of-feb-this-year-already' flag
132	DumpFormat
133	Sync, monitor type, some mode information
	Bit 1 SyncBit
	Bit 2 top bit of mode
	configuration number
	Bits 3 – 4 monitor type
134	Fontsize
135	ADFS use
138	Set *CAT format
140	Set *EX format
142	Twin's CMOS byte
143	Screen size in units of 8K on A1s
144	RAM disc size in units of 8K
145	System heap size
146	RMA size
147	Sprite size
148	SoundDefault parameters
149	For the BASIC Editor
153	Printer server name
158	File server name
173	CMOS used by *Unplug for ROM modules
177	Bits for unplugged modules
181	Wild card CMOS
185	Configured language
186	Configured country
187	VFS CMOS
239	One byte for CMOS RAM checksum; not used currently

If you want to use one or more bytes in an application program, you should request an allocation in writing from Acorn Computers.

Two OS_Bytes are provided for reading and writing the CMOS RAM. They are as follows:

OS_Byte &A1 (161) – Read battery-backed RAM

On entry: R1 = RAM location

On exit: R2 = contents of RAM location

This call provides read access to any of the locations in the battery-backed RAM. For example, this call may be used by a module to read a default configuration parameter. Moreover, this parameter could be examined by the user using the *STATUS command, if the module provides a suitable entry in its command decoding table. See the chapter MODULES for more details.

OS_Byte &A2 (162) – Write battery-backed RAM

On entry: R1 = RAM location
R2 = value to be written

On exit: R2 is undefined

This call provides write access to any of the locations in the battery backed RAM with the exception of location zero, which is protected.

HEAP MANAGER

The OS contains a heap management system. This is used by the operating system to allocate space within the relocatable module area and also to maintain the system heap. A heap is just an area of memory from which bytes may be allocated, then deallocated for later use. An area can also be reallocated, meaning that its size changes.

The heap manager is also available to the user. You provide an area of memory which is to be used for the heap, which can be any size you require. Then, at the start of this area, the heap manager sets up the heap descriptor, which is a block

containing information on the limits of the heap, etc. This descriptor is updated by the heap manager when necessary.

When a block within this heap is required, a request is made to the heap manager, which returns a pointer to a suitable block of memory. The heap manager keeps a record of the total amount of memory which is free in the heap and the largest individual block which is available.

The heap management system does not provide garbage collection and will never attempt to move a block within the heap, since it has no knowledge of whether the block contains pointers that need to be relocated, or whether there are any pointers to the block which need updating. Hence, unless an area of contiguous free space of the size requested is available, a request for a block will fail.

The heap manager SWI call

OS_Heap &1D (29) – Heap manager

The heap manager provides SWI OS_Heap (&1D) to permit use of all its features:

On entry: R0 = reason code
Rn depends on R0, as described below

On exit: V is set if there was an error

As usual, an error causes the error handler to be called if OS_Heap is used. For the call to return with V set and an error pointer in R0, XOS_Heap is used.

The particular action depends on the value of R0 as follows:

R0 = 0 Initialise heap

On entry: R1 points to heap descriptor block to initialise
R3 = size of heap

On exit: V is set if heap could not be initialised

This call takes the area passed for the heap descriptor and places in it the information necessary to transform it into a valid descriptor. The heap is then ready for use. The value given for R1 must be word-aligned and less than 32Mbytes (ie must point to an area of logical RAM). R3 must be a multiple of four and less than 16Mbytes.

R0 = 1 Describe Heap

On entry: R1 = heap descriptor pointer

On exit: R2 = largest available block size
R3 = total free
V is set if heap is invalid

This call returns information on the space available in the heap. V is set if the heap is invalid. This may be for any of the following reasons:

- the heap descriptor is corrupt
- the information within the heap is not sensible
- R1 is not a heap descriptor pointer.

R0 = 2 Get heap block

On entry: R1 = heap descriptor pointer
R3 = size wanted

On exit: R2 = pointer to claimed block or 0 if allocation failed
V is set if allocation failed

This allocates a block from the heap. V is set if the allocation failed which may be for any of the following reasons:

- there is not a large enough block left in the heap
- the heap has been corrupted
- R1 is a nonsensical value.

R0 = 3 Free heap block

On entry: R1 = heap description pointer
R2 = block pointer

On exit: V is set if deallocation is not successful

This checks that the pointer given refers to an allocated block in the heap, and deallocates it. Deallocation tries to join free blocks together if at all possible, but if the block being freed is not adjacent to any other free block it is just added to the list of free blocks. V is set if the deallocation failed which may be because R1 was invalid, the heap descriptor or heap was corrupted or R2 did not point to an allocated block in the heap.

R0 = 4 Extend heap block

On entry: R1 = heap descriptor pointer
R2 = block pointer
R3 = size to change by (signed 32-bit number)

On exit: R2 = new block pointer

This attempts to enlarge or shrink the given block in its current position if possible, or, if this is not possible, by reallocating and copying it. Note that if the block has to be moved, it is your responsibility to note this (by the fact that R2 has been altered), and to perform any necessary relocation of data within the block.

R0 = 5 Extend heap

On entry: R1 = heap descriptor pointer
R3 = size to change by (signed 32-bit number)

On exit: --

This updates the heap size information to take account of the new size. V is set if it cannot shrink far enough, because of data that has already been allocated.

Internal format of the heap

A description of the structure used by the heap manager is given below. It should be noted that this structure is not guaranteed to be preserved between releases of the software and should not be relied upon. It is given purely for advanced readers who may want to interpret the current state of the heap when testing and debugging their own code.

The heap descriptor is a block of four words:

&00	Special heap word
&04	Free list offset
&08	Heap base offset
&0C	Heap end offset

offsets from this word

✓

✓

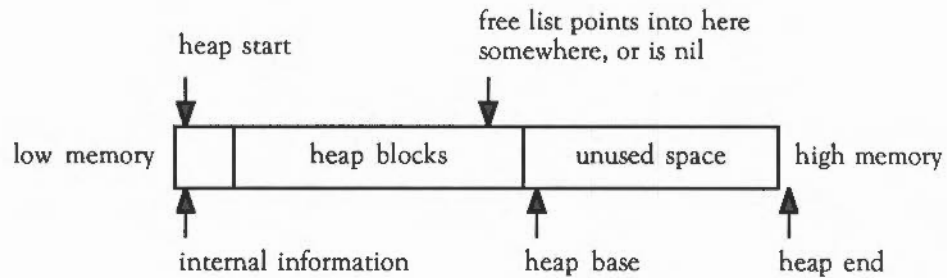
The 'special' heap word contains a pattern which distinguishes correct heap descriptors. The pattern is made up of the characters 'Heap' – which is &70616548 in hex.

All other words are offsets into the heap. This means that the heap is relocatable unless you place non-relocatable information in it.

The free list offset is an offset to the first free block in the heap, or zero if there are no free blocks. If the word is non-zero, the first free block is at address:

heap start + free list offset + 4

The other entries are offsets from the start of the heap which refer to boundaries within the heap structure. The heap is delimited as follows:



Blocks in the free list have information in the first two words as follows:

- Word 0 is the link to the next free block or 0 if at the end
- Word 1 is the size of this block (including these two words).

Allocated blocks start with a word which holds the size of the allocated block. The pointer returned by SWI OS_Heap when a block is allocated actually points to the second word which is the start of the memory available.

Allocation forces the block size to be at least eight to ensure that it can contain the information necessary to free it. Therefore, the minimum size of area that can be initialised is 24 bytes (16 for the fixed information and 8 for a block).

MISCELLANEOUS MEMORY SWIs

This section brings together three SWIs which are connected with memory management.

OS_UpdateMEMC &1A (26)

On entry: R0 = new bits in field
R1 = field mask

On exit: previous state of MEMC register

The memory controller (MEMC) chip is a write-only device. The operating system maintains a software copy of its current state and OS_UpdateMEMC updates MEMC from the software state. To allow the programming of individual bits the call takes a field and a mask. The new MEMC value is:

```
newMemC := (oldMEMC AND NOT R1) OR (R0 AND R1)
R0      := oldMEMC
```

So to read the contents without altering them, R1 and R2 should both be zero. To set them to 'n', R1=&FFFFFFFF and R2=n.

OS_ChangeDynamicArea &2A (42)

On entry: R0 = area to alter
R1 = amount to move

On exit: R1 = bytes moved
V is set if not all bytes moved, or application workspace is being used (ie an application is active)

OS_ChangeDynamicArea allows the system heap or RMA to be altered in size by removing or adding workspace form the application workspace.

The area to be altered depends on R0 as follows:

```
R0 = 0    alter system heap
R0 = 1    alter RMA
```

The amount to move is given by the sign and magnitude of R1:

```
+ve      means enlarge RMA/heap
-ve      means give space back to application workspace
```

Note that this cannot be used while the application work area is being used, eg when a language is active. An attempt to do so will result in a Memory in use error. (In fact, when this call is made, the OS passes a service call round to modules, which can

veto the change if they can't handle it correctly. See the chapter **MODULES** for more details.)

```
Example: MOV R0, #1           ; select RMA
           MOV R1, #&80000000 ; largest negative number
           SWI OS_ChangeDynamicArea ; shrink RMA as far as possible
```

OS_ValidateAddress &3A (58)

On entry: R0 = minimum address
R1 = maximum address

On exit: C = 0 if OK, 1 otherwise

SWI OS_ValidateAddress checks the address range between R0 and R1 minus 1 to see if they are valid. Validity in this case means that the addresses are in logical RAM (0 – 32M) and have a mapping into physical RAM throughout the range.

THE PROGRAM ENVIRONMENT

This chapter describes how the application program, which is currently running, can read and set important aspects of its 'environment', such as the highest location available to it, the addresses of various 'handlers' and so on. It also summarises the various ways in which you can execute a program, and what the program can and can't do while running.

Operating system variables are also described in detail in this chapter.

RUNNING AN APPLICATION

There are several ways in which a program becomes the current application. At present, Arthur is a single tasking environment, and only one foreground (ie non-interrupt) program may be active at once. However, it is possible to run an application from another program, and have the first program restart when the application terminates.

Here are the ways in which a program can become the current application:

- By *RMRUNning the program from the filing system
- By OS_Module 'Enter'ing (for modules)
- By *RUNning the program from the filing system
- By executing the program using *GO

The first and second cases are really the same thing. When a file is *RMRUN, it is loaded into the relocatable module area. Its initialisation code is called, so that it can claim workspace etc, then its start code is called.

A module can also call its own start entry point (using OS_Module) if it wants to become the current application. BASIC is an example of this. The *BASIC command is recognised by the OS using the BASIC module's * command table. The OS calls the routine which handles the *BASIC command, and this routine calls OS_Module with the reason code 'enter'. See the next chapter for details on calling modules.

The third case applies to files which have no file type, or have type FF8. In the first case, the file is loaded at its load address, then it is started as an application through

its execution address. If the file type is FF8, the file is loaded at address &8000 and started as the application there.

Finally, if you call a machine code program using the *GO command, it becomes the current application. (This implies that you shouldn't use *GO to call RAM-based routines from a language, as the routine can't return – R14 contains no return address at this point.)

In all of these cases, the program is called in user mode, with interrupts enabled. Where a module is called, R12 points to the module's private word.

The running application has the application workspace available to it. This starts at &8000 and extends to an upper limit which depends on:

- the amount of RAM in the machine
- the number of active modules
- various configuration parameters, etc.

See the previous chapter for an example of how the application area's size is determined using these factors.

Application workspace is always contiguous. It is guaranteed not to change while the application is active. (Actually, if the application is a module, it can decide whether to let the workspace change or not by responding appropriately to a service call.)

Within the Arthur Supervisor prompt, the size of the RMA is dynamically alterable, and any changes caused by, for example, the loading of a module, will be reflected in the amount of RAM available to the next application to run.

LEAVING AN APPLICATION

Before describing the calls which control the application program's environment, it is worth explaining how to leave an application. In general, a simple 'return from subroutine' using MOV PC,R14 won't suffice. Instead, you should use a routine called OS_Exit. This passes control back to a well-defined place, which defaults to the supervisor * prompt, but could equally be a location in the previous application.

OS_Exit &11 (17)

On entry: R0 = pointer to error block
 R1 = "ABEX" (&58454241) if return code is to be set
 R2 = return code

On exit: never returns

When OS_Exit is called, control returns to the most recent exit handler address (see below). The BASIC statement QUIT performs an OS_Exit. Before executing OS_Exit, however, you should restore any of the handlers changed in starting the application.

If the exiting program wishes to return with a result code, it must set R1 to the hex value shown above, and R2 to the desired value. Non-error results must be in the range 0 to the value of the variable Sys\$RCLimit. The return value is assigned to the variable Sys\$ReturnCode, which can be interrogated by any program using OS_ReadVarValue.

To return with an error, exit with a value less than zero or greater than Sys\$RCLimit (having restored the previous error handler, as indicated above). This gives the error Return code limit exceeded (&1E2), but still sets the variable to the required value.

Example:

```

MOV R2, #errorResultCode
LDR R1, returnString
SWI "OS_Exit"
.returnString
EQUUS "ABEX"

```

THE ENVIRONMENT SWIs

This section lists the SWIs which are used to control the various handlers and other addresses used by the system. Note that the error and event handlers are called by the default owners of the respective vectors. These are used if no routine on the vector intercepts it (which should never happen).

OS_Control &OF (15) – Set/read handler addresses

On entry: R0 = address of error handler
R1 = pointer to buffer for the error handler
R2 = address of escape state change handler
R3 = address of event handler

On exit: R0 = previous error handler address
R1 = previous buffer address
R2 = previous escape routine address
R3 = previous event handler address

OS_Control sets some of the exception handlers. The addresses of the error handler, error handler buffer, escape state change handler and event handler are passed in R0 – R3. Zero for any of these means no change, ie a read-only operation.

When a handler is called, only registers R10 – R12 should be relied upon to contain meaningful values.

The error handler is called after any error has been generated. It is called by the default owner of the error vector; thus any routines using this vector should always 'pass it on'. Continuing after an error is not generally recommended. You should always use the X form SWIs if you wish to stay in control even when an error occurs. Note that if the error handler is set up using OS_ChangeEnvironment, the workspace pointer is passed in R0, not R12 as usual.

The error buffer (whose address should be set along with the handler address) contains the following:

R1+0 – R1+3	PC when error occurred
R1+4 – R1+7	Error number provided with the error.
R1+8 ...	Error string, terminated with a 0

The escape handler is called when the escape status changes. The routine is entered with the following:

R11 bit 6 is escape state. 1 => escape condition.

R12 does not contain 1
R13 is a full, descending stack pointer

To continue after an escape, the handler should reload the PC with the contents of R14. If R12 contains 1 on return then CallBack will be used. CallBack should be used if the handler wishes to do a lot of work or wishes to enable interrupts or to execute an OS_GenerateError. Typically (eg BASIC), the handler will set an internal flag which is checked by the foreground program.

The event handler is called when an event occurs, and is called by the default owner of EventV with the following:

R0 event reason code
R1... parameters according to event code
R12 does not contain 1
R13 is the IRQ handler routine stack pointer

To continue after an event, the handler should reload the PC with the contents of R14. If R12 contains 1 on return then the CallBack handler is used.

You should take care not to corrupt R14_SVC during handler code. This implies saving it on the stack if you use SWIs. See the section **Interrupts** in the chapter **Fundamental operating system concepts** for details.

The handlers are initialised by the OS within the supervisor * prompt. The error handler reports the error message and number. The escape handler and event handler do nothing. BASIC sets up its own error handler and escape handler and leaves the event handler alone.

Note that the call OS_ChangeEnvironment provides more control over the handlers than this call, and should be used in preference. (OS_Control actually uses OS_ChangeEnvironment.)


```

Example:  ADR    R0, errorHandler
          MOV    R1, #work+&300
          MOV    R2, #0
          MOV    R3, #0
          SWI    "OS_Control"      ; control errors

```

OS_GetEnv &10 (16) – Read environment parameters

On entry: –

On exit: R0 = address of the * command string
 R1 = permitted RAM limit (ie highest address available)
 R2 = address of 5 bytes – the time the program started

OS_GetEnv reads the program environment.

The value returned in R0 is the address of the * command which caused the application to start. It skips spaces and *, but not the command name which it uses to read command line parameters. (Note, however, that the string pointed at is corrupted by any OS_CLI calls.)

R1 returns the address of the byte above the last one available to the application. From &8000 to this address minus one is available to the application.

BASIC uses &8000 to &86FF as workspace, and from &8700 to the RAM limit for program as variable storage. HIMEM defaults to the value returned in R1, and BASIC's stack starts from the last word of application workspace and grows down.

The five bytes pointed to by R2 give the time in centi-seconds since 1899, and can be converted to a string using, for example, OS_ConvertStandardDateAndTime.

```

Example:  SWI    OS_GetEnv
          SWI    OS_Write0          ;write environment string to terminal

```

OS_SetEnv &12 (18) – Set environment parameters

On entry: R0 = address of the routine for OS_Exit above to go to
 R1 = address of the end of memory limit for OS_GetEnv to read
 R2, R3 are undefined
 R4 = address of routine to handle undefined instructions
 R5 = address of routine to handle prefetch abort
 R6 = address of routine to handle data abort
 R7 = address of routine to handle address exception

On exit: R0 – R7 = previous values of the above

OS_SetEnv sets the program environment. Most applications will not need to change these handlers. The debugger is an example of one that would.

Undefined, abort and exception handlers are initialised in the operating system supervisor (* prompt). Giving zero instead of an address indicates no change, so the previous value can be read. The addresses set by this instruction are simply stored in system workspace locations where they can be accessed by code at the hardware vectors. For further details, see section **Hardware vectors** in the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS** for details. All of the default handlers simply generate error messages.

– *Note:* the call OS_ChangeEnvironment provides a superset of the facilities that this call provides, and should be used in preference.

Example:

```

ADR    R0,ExitRout
MOV    R1,#&10000
MOV    R4,#0
MOV    R5,#0
MOV    R6,#0
MOV    R7,#0
SWI    "OS_SetEnv"          ; simulate a small machine

```

OS_CallBack &15 (21) – Set-up CallBack handler

On entry: R0 = address of the register save block
R1 = address of the CallBack handler

On exit: previous values of the above

OS_CallBack sets up the address of the CallBack handler and the register save block, zero for either value meaning no change.

This CallBack code provides a method of calling a routine on exit from the operating system. As the thread of control leaves the system, instead of returning to the original caller, the OS saves the registers in a save block and a different routine is entered. Control can be resumed later by reloading from the register save block.

Entry to the CallBack code is performed whenever the OS's internal CallBack flag is set. This may be done explicitly, by OS_SetCallBack (see below). Alternatively, an escape or event handler returning with 1 in R12 will set the CallBack flag. (See OS_Control).

Having set the flag, the CallBack routine is then entered when the system next exits to user mode code with interrupts enabled. Whenever the CallBack handler is called, the CallBack request flag is cleared.

The CallBack code is called in IRQ or supervisor mode with interrupts disabled. The PC stored in the save block will be a user mode PC with interrupts enabled. Note that if the currently active program has interrupts disabled or is running in supervisor mode, CallBack is not used.

In the simple case the CallBack routine should be exited by:

```
ADR    R14, saveblock      ; get address of saved registers
LDMIA  R14, {R0 - R14}^    ; load user registers from block
LDR    R14, [R14, #15*4]   ; load user R14 into SVC R14
MOVS   PC, R14             ; return using it
```

```

Example:      ADR    R0,saveblock
              ADR    R1,nullroutine
              SWI    "OS_CallBack"
              .....
.nullroutine ADR    R14,saveblock
              LDMIA R14,{R0 - R14}^
              LDR    R14, [R14,#15*4]
              MOVS   PC, R14

```

- Note: as you might want to call this SWI from IRQ code, you should take the precautions necessary to preserve R14_SVC shown in section **Interrupts** in the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS**.

OS_SetCallBack &1B (27)

On entry: --

On exit: --

OS_SetCallBack sets the CallBack flag and so causes entry to the CallBack handler, set up by OS_CallBack, when the system next exits to user mode code with interrupts enabled (apart from the exit from this SWI). This SWI may be used if the code linked into the system (via a vector or as a SWI handler, etc) is required to do things on exit from the system.

OS_BreakPt &17 (23)

On entry: --

On exit: --

OS_BreakPt forms a break point trap. See OS_BreakCtrl below.

```

Example: SWI    "OS_BreakPt"

```

OS_BreakCtrl &18 (24)

On entry: R0 = address of the register save block; 0 for no change
R1 = address of the control routine; 0 for no change

On exit: R0, R1 = previous values of the above
V is always clear

When OS_BreakPt is executed, all the user mode registers are saved in a block and execution is continued at the break control routine which is set up by OS_BreakCtrl. The saved registers are only guaranteed to be correct for user mode.

The default handler prints the register contents from the save block and enters the Arthur Supervisor.

The handler is entered in SVC mode. To restore user registers from the save block and return, the code given above for returning from a Callback routine may be used.

See also OS_ChangeEnvironment for an alternative way of setting up the break point handler, so that the handler's R12 is set up.

Example:

```
ADR    R0,saveBlock
ADR    R1,controlRoutine
SWI    "OS_BreakCtrl"
```

OS_UnusedSWI &19 (25)

On entry: R0 = address of the unused SWI handler; 0 for no change

On exit: R0 = previous value of the above

If the OS can't decode the number of a SWI into one which it supports directly, or which can be handled by a module, the OS calls the vector SWIV. This allows a user routine on that vector to try to deal with the SWI. If there is no such routine, or the one(s) that is present passes the call on, then the Unused SWI handler is called.

The entry conditions for the handler are:

- Entered in supervisor mode
- Interrupts are disabled
- R11 contains the SWI number (Bit 17 clear)
- R13 is the SVC stack pointer
- R14 is the user PC with V cleared
- R10, R11 and R12 are stacked and are free for your own use.

This handler may also be set up using `OS_ChangeEnvironment`.

`OS_ChangeEnvironment` &40 (64)

On entry: R0 = handler number
R1 = new address, or 0 for no change
R2 = R12 with which to call the routine, or 0 for no change
R3 = buffer pointer, if appropriate

On exit: R1 = previous address
R2 = previous R12
R3 = previous buffer pointer

`OS_ChangeEnvironment` is a single routine which performs the actions of `OS_Control`, `OS_SetEnv`, `OS_CallBack`, `OS_BreakCtrl`, and `OS_UnusedSWI`. In fact, all of those routines use this call.

On entry, R0 contains a code which determines which particular handler's address is to be set up. The new address is passed in R1. R0 also determines whether R2 and R3 are relevant or not. This is summarised in the table below.

R0	Handler	R2	R3
0	MemoryLimit	Ignored	Ignored
1	Undefined ins.	Ignored	Ignored
2	Prefetch abort	Ignored	Ignored
3	Data abort	Ignored	Ignored
4	Address exception	Ignored	Ignored
5	Other exception	Ignored	Ignored
6	Error	R0 when called	Error buffer address
7	CallBack	R12 when called	Register buffer address
8	BreakPoint	R12 when called	Register buffer address
9	Escape	R12 when called	Ignored
10	Event	R12 when called	Ignored
11	Exit	R12 when called	Ignored
12	Unused SWI	R12 when called	Ignored
13	Exception regs.	Ignored	Ignored

'Other exceptions' (handler 5) is for future expansion.

Handler 13 sets the address of the area in memory where the registers are dumped when one of the exceptions (1 – 5) occurs, if the default handlers are used.

Note that in order to perform its function, OS_ChangeEnvironment vectors through ChangeEnvironmentV. A routine linked onto this vector can stop the change from happening by setting R1 (and if appropriate R2, R3) to zero and passing the call on. See also **Software** vectors in the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS**.

In new programs, you should always use this call in preference to the earlier ones.

OPERATING SYSTEM VARIABLES

The variables, maintained by the operating system in the system heap, provide a convenient way by which programs can communicate. OS_Exit (above) shows how the variable Sys\$ResultCode is set up on exit from a program while the command line interface to handling variables is described in the chapter **COMMAND LINE INTERPRETER**. This section describes the SWIs which handle the creation, setting, reading and deletion of variables.

Variables are accessed by their textual name. The name may contain any non-space, non-control character. When a variable is created, the case of the letters is preserved. However, when names are looked up, the case is ignored. Variable names act much like filenames in this respect.

You should avoid the use of wholly numeric names for variables, such as 123, as this causes difficulties when OS_GSTrans is used to look up a variable's contents. In particular, OS_GSRead will always take <123> to mean the ASCII code 123, and will not attempt to look up the name as a variable.

There are several types of variable. String variables contain ASCII characters only, which are returned when the string is read. Integer variables are four-byte signed integers. Macros are like string variables, but when they are read, the string is OS_GSTransed. This means that if the macro contains references to variables or other OS_GSReadable items, the appropriate translation takes place whenever the variable is accessed.

A classic example of using a macro is to set the Arthur supervisor prompt Cli\$Prompt to the current time using:

```
*SETMACRO cli$prompt <sys$time><&20>
```

Every time the prompt is displayed, it shows the current time, followed by a space.

The final type of variable is machine code routines. A routine is called whenever the variable is to be read, and another when it is set. This allows great flexibility in the way in which such variables behave. For example, you could make a variable directly control a CMOS RAM location using this technique.

OS_ReadVarVal &23 (35) – Read a variable's value

On entry: R0 = pointer to name, may be wildcarded (* and #)
R1 = pointer to buffer
R2 = maximum length of buffer
R3 = name pointer (or 0 for first call).
R4 = 3 if an expanded string is to be returned

On exit: R2 = number of bytes read
R3 = new name pointer, string is null-terminated
R4 = type of variable (string, number or macro)
V is set if can't find (R2=0), or buffer overflowed

let preserved

string if R4 was 3

OS_ReadVarVal reads a variable and returns its value and its type. On entry, R3 should be 0 the first time the call is made for a wildcarded name, and thereafter preserved from the previous call. This enables all matches of a wildcarded name to be found. On exit, R3 points to the name of the variable found. The XOS_ReadVarVal form of the call should be used if you don't want an error to occur after the last name has been found.

You can call XOS_ReadVarVal to check for the existence of a variable by setting R2 to a value less than zero (bit 31 set) on entry. If it is still negative on exit, the variable exists; if it is zero, the variable does not exist.

The type of the variable read is returned in R4 as follows:

Value	Type
VarType_String (0)	String
VarType_Number (1)	4 byte (signed) integer
VarType_Macro (2)	Macro

R4, if set to 3 on entry, indicates that a suitable conversion to a string should be performed. String variables are unaltered, numbers are converted to (signed) decimal strings, and macros are OS_GStransed.

If R4 isn't 3 on entry, the un-OS_GSTransed version of a macro is returned, and the four-byte binary of a number is returned.

ie NOT the actual value

```

Example: ;Print all sys$ variable names
          ADR   R1, valBuffer      ;Buffer to place value
          MOV   R3, #0             ;Initial context

.loop
          ADR   R0, strName        ;Wildcarded name to find
          MOV   R2, #bufferLen     ;Length of value buffer
          SWI   "XOS_ReadVarVal"   ;Non-error reporting one
          MOVVSS PC,R14            ;Return and clear V
          MOV   R0, R3             ;Get address of name
          SWI   "OS_Write0"        ;Print it
          SWI   "OS_NewLine"      ;and new line
          B     loop              ;again
          ....
.strName  EQU   "SYS$*" + CHR$0
    
```

OS_SetVarVal &24 (36) – Set/create a variable

On entry: R0 = pointer to name (can be wildcarded for update/delete)
 R1 = pointer to value
 R2 = length of value. Negative means destroy the variable
 R3 = name pointer (or 0 for first call)
 R4 = type (see below)

On exit: R3 = new context pointer
 R4 = type created if expression is evaluated
 V is set if an error occurred (X version)

OS_SetVarVal either creates, updates or destroys a variable. The name may be terminated by any character whose ASCII value is 32 or less and may be wildcarded if it is to be updated or deleted (ie if it already exists).

The pointer to the value to be assigned in the case of create/update is given by R1. If it is a string then it must be terminated by a linefeed (ASCII 10) or carriage return

(ASCII 13). The interpretation of the value depends on the type given in R4 as follows:

VarType_String	(0)	OS_GSTrans the given value
VarType_Number	(1)	Value is a 4 byte (signed) integer
VarType_Macro	(2)	Copy value (may be OS_GSTransed on use)
VarType_Expanded	(3)	The value is a string which should be evaluated as an expression using OS_EvaluateExpression, and assigned to a number or string variable, depending on the expression type
VarType_Code	(16)	Special case (see below)

If the call is successful, R3 is updated to point to the new context so allowing the next match of a wildcarded name to be obtained on a subsequent call. R4 returns the type created if an expression was evaluated (ie if R4 was 3 on entry).

R2 must be negative on entry to delete a variable. Also, to delete a type-16 variable, R4 should contain 16 on entry.

When R4 is set to 16 on entry (and R2 \geq 0) a code variable may be created. In this case R1 is the pointer to the code fragment associated with the variable, and R2 is the length of the code fragment. This code must be word-aligned and takes the following format:

Offset	Contents
0	Branch instruction to entry point for write operation
4	Entry point for read operation
8...	Body of code
...	...

The entry for the write operation is called whenever the variable is to be set. It is called as follows:

On entry: R1 = pointer to the value to be used
R2 = length of value

On exit: R1, R2, R4, R10 – R12 may be corrupted

PELF

The entry for the read operation is called whenever the variable is to be read. It is called as follows:

On entry: -

On exit: R0 = pointer to value
R1 may be corrupted
R2 = length of value

Both entries are called in SVC mode. Therefore if any SWIs are used, R14 must be saved on the stack so that it does not become corrupted. Below is a complete example of a program to create a variable called Mode. The read action is to return the current display mode, and the write action to set the mode.

```
.start      ADR    R0, varName      ;Pointer to the name
            ADR    R1, code       ;Start of code body
            MOV    R2, #endCode-code ;Length of code body
            MOV    R3, #0         ;Context pointer
            MOV    R4, #&10      ;'special' type
            SWI   "OS_SetVarVal"  ;Create it
            MOV    PC, R14       ;Return

.code       B      writeCode     ;Branch to write code
.readCode  STMFD  R13!, {R14}    ;Save return address
            MOV    R0, #&87      ;OS_Byte read mode number
            SWI   "XOS_Byte"
            MOV    R0, R2        ;Mode in R0 for conversion
            ADR    R1, buffer     ;Buffer for ASCII conversion
            MOV    R2, #4        ;Max len of buffer
            SWI   "XOS_BinaryToDecimal"
            MOV    R0, R1        ;Pointer in R0
            ;length already in R2
            LDMFD R13!, {PC}     ;Return
```

```

.writeCode   STMFD R13!, {R14}           ;Save return address
             SWI  "XOS_ReadUnsigned"    ;R1 set correctly already
             SWI  &100+22              ;VDU mode change
             MOV  R0,R2                 ;Get integer read in R0
             SWI  "XOS_WriteC"         ;Do mode change
             LDMFD R13!, {PC}          ;Return

.buffer      EQU  0                    ;Buffer for string conversion
.endCode

.varName     EQU  "Mode "              ;Name of variable

```

The routine at 'start' creates the variable. Obviously as the code body is copied into the system heap, it must be position independent. The two routines 'readCode' and 'writeCode' are called whenever an access to the variable is made. For example, a *SET Mode command will call the write code entry, and *SHOW sys\$mode or *ECHO <Mode> will call the read entry.

Notice that in the body of the code variable, only XOS_ SWIs are used. This is because it is important that errors are not generated when the read or write code executes. A more rigorous version of the code above would check V after each SWI and return if it was set.

- Note: when a function key is input, the appropriate variable key\$n is read using OS_ReadVarVal. Therefore by creating your own code variables with these names, you can cause the reading of a function key to cause a routine to be called instead of just a string being read.

OS_SetVarVal can return the following errors:

- Bad name Wildcards/control characters in name when creating
- Bad string OS_GSTrans unable to translate string
- Bad macro value Control characters in the value string (R1)
- Bad expression Expression cannot be evaluated

- Variable not found For deletion or update
- No room for variable Not enough room to create/update it (system heap full)
- Variable value too long Variables are limited to 256 bytes
- Bad variable type

SUMMARY OF EXECUTION MODES

This section summarises the ways in which control may be passed to user's code, apart from the case of the code being the current application, which is described above.

Vectors are called in SVC or IRQ mode – it is difficult to predict which sometimes. In view of this, you should assume nothing about the mode you are in. If you want to execute a SWI, you should enter SVC mode explicitly, push R14_SVC, call the SWI, restore R14_SVC and finally restore the processor mode. The code to do this is shown in section **Interrupts** in the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS**.

Module entries

If a program is a module loaded into the RMA, there are several ways in which it can be called. At the start of the module, there are several offsets which are used by the OS to call the module at various times. These include an initialisation entry and a service code entry. See the next chapter for a full description.

Additionally, a module may have a SWI and * command table. These are used to enable the OS to pass control automatically to the module if it can handle an unrecognised SWI or command. Also, the OS can recognise a module's *CONFIGURE and *STATUS options for it.

Vectors and handlers

There are many vectors and other 'hooks' to which user code can attach itself (see above and the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS**.) These are usually called in supervisor or IRQ mode, as they are an indirect result of the user executing a SWI, or some interrupt event occurring.

In general, all extensions to the system should be implemented as modules. Therefore the SWI vector should not be used, as SWIs can be passed to modules automatically. Also, the OS_CLI vector should only be used if a module wants to replace totally the command line interpreter. This is not very wise. It can be useful to use CliV if you want to know about every * command that is issued (for debugging purposes, for example). In this case, the vector would always be passed on.

Handlers tend to be taken over by applications to ensure that control returns to a well-defined point in the program. See the example above of the way BASIC uses the escape and error handlers.

Code called by vectors and handlers which are set-up using OS_ChangeEnvironment can always rely on the workspace pointer in R12 being set up. (The exception is the error handler – the workspace pointer is in R0.) Similarly, a full, descending stack pointer will be in R13; this may be the SVC or IRQ stack, depending on what routine is called. Vector code should always take care to obey the appropriate register conventions – preserving register where necessary and passing back results in the correct registers/flags.

Transient programs

A file with type FFC (utility) must contain position independent code. When such a file is *RUN, it is loaded into the RMA and executed. The following entry conditions exist:

On entry: R0 = pointer to command line (OS 0.03 onwards)
R1 = pointer to command tail
R12 = pointer workspace
R13 = pointer to workspace end (stack)
R14 = return address
User mode, interrupts enabled

The workspace is 1024 bytes located directly after the loaded program file. If more is required, it may be allocated from the RMA. The utility should return using MOV PC,R14 (freeing any extra workspace first). It does not become the current application and should not call OS_Exit.

Note that R0 points to the first character of the command name, and R1 points to the first character of the command tail (with spaces skipped). This will be a control character if there were no parameters.

When a utility returns, the space it occupies is freed. Utilities are nestable – you can execute one utility from within another.

Note that utilities are viewed as system extensions. This means that they should only use the X form SWIs, so that the error handler is not called by their actions. Alternatively, the utility can set up its own error handler, as long as it restores the previous value before returning. A utility can return with an error by setting V and pointing R0 at an error block as usual.

- ADFS 209
- ADFS error messages 277
- ADFS intrinsic commands 266
- ADFS SWI calls
 - perform a miscellaneous disc operation 272
 - read free space 277
 - set address of hard disc controller etc. 276
- ADFS_DiscOp 272
- ADFS_Drives 277
- ADFS_FreeSpace 277
- ADFS_HDC 276
- advanced disc filing system 263
- aliases 186
- anti-aliasing palette 509
- application, leaving 334
- ArgsV 24
- argument passing in external procedures 626
- arithmetic instructions 604
- ARM assembler 595
- ARM instruction set 603
- ARM procedure-call standard 623
- Arthur OS 3
- ASCII to binary conversions 403
- assembler 595
- assembly language statements, format 600
- auto-repeat 147
- banks 67
- BASIC assembler 595
- battery-backed RAM 325
- BGetV 24
- Binary to ASCII conversion SWIs 407
- binary to ASCII conversions 407
- BPutV 24
- branching instructions 607
- buffer OS_Byte calls
 - examine buffer status 46
 - flush buffer 44
 - flush selected buffer 45
 - get buffer/mouse status 45
 - get character from buffer 46
 - insert character code into buffer 46
 - insert character into buffer 47
- buffer codes, interpreting 144
- buffer numbers and sizes 43
- buffers 42
- ByteV 23
- ChangeEnvironmentV 29
- character output routines
 - perform a plot command 53
 - print a formatted string 52
 - write a counted string 53
 - write an immediate byte 53
 - write an in-line string 51
 - write an indirect string 51
 - write character 51
 - write newline 52
- character input event 38
- character output 51
- chunk numbers 7
- CLI 183
- CLI parameters, reading 207
- ClV 23
- clock/calendar 396
- CMOS RAM 322
- CnpV 26
- co-ordinate units 506
- command keyword table 381
- command line interpreter 12, 183

- comparisons 606
- condition codes 602
- control codes 69
- conventions 1
- country flag, read 49
- cursor editing 68

- date 391
- date stamping 213
- debugger * commands 567
- debugger 567, 617
- Debugger_Disassemble 572
- defect list 274
- define pointer and mouse parameters 130
- define font 507
- dialogue boxes 444
- directories 210
- disc specifiers 265
- Double Extended Precision (E) 574
- dragging boxes 445

- Econet 585
- Econet conventions 589
- Econet event 40
- environment SWIs
 - read environment parameters 338
 - set environment parameters 339
 - set-up CallBack handler 340
 - set/read handler addresses 336
- error handling, SWI 8
- error numbers 9
- errors, generating 11
- ErrorV 22
- escape condition 147
- escape condition event 39

- event OS_Bytes
 - disable event 36
 - enable event 37
- event dispatcher 550
- event queue 549
- events 36
- EventV 25
- execution modes 351
- expression evaluator 405

- file types 213
- fileswitch 217
- FileV 24
- filing system
 - interface calling conventions 297
 - interface calls 298
 - writing your own 293
- filing systems 209
- floating-point emulator 573
- floating-point instruction set 578
- floating-point status register 576
- font files 490, 513
- font manager 489
- font manager SWIs 492
- font painter 490, 505
- Font_CacheAddress 493
- Font_Caret 497
- Font_CharBBox 500
- Font_ConverttoOS 497
- Font_Converttopoints 498
- Font_CurrentFont 498
- Font_FindCaret 499
- Font_FindCaretJ 504
- Font_FindFont 493
- Font_FutureFont 499
- Font_ListFonts 501
- Font_LoseFont 493

- Font_Paint 496
- Font_ReadDefn 494
- Font_ReadInfo 494
- Font_ReadScaleFactors 500
- Font_ReadThresholds 502
- Font_SetFont 498
- Font_SetFontColour 501
- Font_SetPalette 502
- Font_SetScaleFactors 500
- Font_SetThresholds 504
- Font_StringBBox 505
- Font_StringWidth 494
- fonts, accessing 491
- format of the heap 329
- FSCV 25
- FSEntry_Args 301
- FSEntry_Close 304
- FSEntry_File 305
- FSEntry_Func 309
- FSEntry_GetBytes 300
- FSEntry_Open 298
- function keys 145
- function-key codes 166

- GBPBV 24

- hardware vectors 30
- heap descriptor 329
- heap manager 325
- heap manager SWI call 326
- help keyword table 381

- IEEE Double Precision (D) 574
- IEEE Single Precision (S) 574
- input buffer event 38
- input routines 139
 - read character 139
 - read key with time limit 139
 - read line from input stream to
 - memory 141, 142
- input stream OS_Bytes
 - read input source 139
 - specify input stream 138
- input streams 137
- instruction set 603
- InsV 25
- interrupts 31
- interrupts
 - devices handled under 3
 - disabling 35
 - intercepting 32
- interval timer event 39
- IrqV 23

- key up/down event 41
- keyboard input 443
- keyboard interrupts 143
- keyboard OS_Byte calls
 - acknowledge escape condition 157
 - clear escape condition 157
 - cursor key status 148
 - keyboard scan 156
 - keyboard scan from 16 decimal 156
 - read last break type 172
 - read/write Break and Escape effect
 - 162
 - read/write Break key actions 172
 - read/write Ctrl function key
 - interpretation 167
 - read/write Ctrl Shift function key
 - interpretation 167
 - read/write cursor key status 149
 - read/write escape character 165
 - read/write escape effects 169

- read/write Escape key status 169
- read/write function key interpretation 167
- read/write interpretation of input values &C0 – &CF 165
- read/write interpretation of input values &D0 – &DF 165
- read/write interpretation of input values &E0 – &EF 165
- read/write interpretation of input values &F0 – &FF 165
- read/write keyboard auto-repeat delay 150
- read/write keyboard auto-repeat rate 151
- read/write keyboard disable flag 162
- read/write keyboard semaphore 161
- read/write keyboard status byte 163
- read/write length of function key string 164
- read/write numeric keypad interpretation 170
- read/write Shift function key interpretation 167
- read/write Tab key code 164
- reflect keyboard status in LEDs 151
- reset function keys 151
- scan a for a particular key 158
- set effect of Shift Ctrl on numeric keypad 173
- set escape condition 157
- write keyboard auto-repeat delay 149
- write keyboard auto-repeat rate 150
- write keys pressed information 152
- keyboard scanning 146
- keyboard SWI calls 174
- layout of windows 439
- Level 1 sound channel handler 531
- Level 2 sound scheduler 534
- line input 140
- linker 613
- linker keywords 616
- linker symbols 615
- load/save instructions 608
- logical instructions 604
- logical memory map 318
- logical RAM 317
- memory management 317
- memory protection 320
- memory SWIs 330
- metrics files format 513
- miscellaneous OS_Byte calls
 - display OS version information 48
 - read country flag 49
 - read/write user flag 48
 - write user flag 48
- modes, screen 65
- module * commands 356
- module code, errors in 365
- module header format 366
- module help string 380
- module SWI calls
 - issue module service call 364
 - perform a module operation 358
- module title string 380
- module workspace 365
- module, writing 365
- modules 355
- mouse and pointer 128
- mouse button event 40
- mouse buttons 443

- mouse/pointer OS_Byte calls
 - get buffer/mouse status 129
 - select pointer / activate mouse 128
- mouse/pointer OS_Word call
 - define pointer and mouse parameters 130
- MouseV (&1A) 28
- multiply instructions 607

- NetFS 209
- NetFS intrinsic commands 283
- netprint 290
- network filing system 283
- network filing system
 - configuration 285
 - interfaces 286
- network printing 290
- non-volatile memory 322
- Num Lock, effect of 171
- number conversions 403
- number conversions, SWI 409

- OpenV 24
- operating system commands 189
- operating-system variable calls
 - read a variable's value 346
 - set/create a variable 347
- operating-system variables 345
- OPT assembler directive 599
- OS filing system commands 217
- OS filing system SWI calls
 - check for end of file 231
 - filing system control 254
 - open or close a file for byte access 242
 - perform action on whole file 233
 - read or write arguments for an open file 252
 - read single byte from an open file 251
 - read/write a group of bytes from/to an open file 244
 - read/write boot option 232
 - write filing system options 232
 - write single byte to an open file 251
- OS table 187
- OS vectors
 - 100Hz pacemaker vector 29
 - buffer insert vector 25
 - buffer remove vector 25
 - command line interpreter vector 23
 - count/purge buffer vector 26
 - error vector 22
 - event vector 25
 - file arguments read/write vector 24
 - file byte block get/put vector 24
 - file byte put vector 24
 - file byte read vector 24
 - file open vector 24
 - file read/write vector 24
 - filing system control vector 25
 - mouse vector 28
 - OS_Byte indirection vector 23
 - OS_Word indirection vector 24
 - read character vector 23
 - read line vector 25
 - unknown interrupt vector 23
 - unknown SWI vector 27
 - unknown VDU 25 vector 27
 - unknown VDU23 vector 27
 - VDU extension vector 28
 - warning of change in environment 29
 - warning vector 29
 - write character vector 23
- OS version information, display 48
- OS_Args 252

OS_BGet 251	OS_GSTrans 414
OS_BinaryToDecimal 407	OS_Heap 326
OS_BPut 251	OS_InstallKeyHandler 174
OS_BreakCtrl 342	OS_Module 358
OS_BreakPt 341	OS_Mouse 134
OS_Byte 13	OS_NewLine 52
OS_Byte calls	OS_Plot 53
buffers 44	OS_PrettyPrint 52
index of 643	OS_ReadC 139
mouse/pointer 128	OS_ReadEscapeState 174
RS423 176	OS_ReadLine 141
VDU 104	OS_ReadModeVariable 124
OS_CallAfter 394	OS_ReadMonotonicTime 394
OS_CallAVector 18	OS_ReadPalette 121
OS_CallBack 340	OS_ReadPoint 124
OS_CallEvery 395	OS_ReadUnsigned 403
OS_ChangeDynamicArea 331	OS_ReadVarVal 346
OS_ChangeEnvironment 343	OS_ReadVduVariables 122
OS_CheckModeValid 127	OS_Release 18
OS_Claim 17	OS_RemoveCursors 126
OS_ClaimScreenMemory 127, 322	OS_RemoveTickerEvent 395
OS_CLI 184	OS_RestoreCursors 127
OS_Control 336	OS_ServiceCall 364
OS_ConvertDateAndTime 401	OS_SetCallBack 341
OS_ConvertStandardDateAndTime 400	OS_SetEnv 339
OS_EvaluateExpression 404	OS_SetVarVal 347
OS_Exit 335	OS_SetVarVal errors 350
OS_File 233	OS_SpriteOp 422
OS_Find 242	OS_SubstituteArgs 414
OS_FSControl 294	OS_SWINumberFromString 410
OS_FSControl 254	OS_SWINumberToString 410
OS_GBPB 244	OS_UnusedSWI 342
OS_GenerateError 11	OS_UpCall 19
OS_GenerateEvent 37	OS_UpdateMEMC 330
OS_GetEnv 338	OS_ValidateAddress 332
OS_GSInit 411	OS_Word 13
OS_GSRead 412	

- OS_Word calls
 - mouse/pointer 130
 - index of 646
 - VDU 116
- OS_Write0 51
- OS_WriteC 51
- OS_WriteC, using 69
- OS_WriteI 53
- OS_WriteN 53
- OS_WriteS 51
- Output stream OS_Bytes
 - read printer driver type 61
 - read/write *SPOOL file handle 63
 - read/write character destination status 55
 - read/write NOIGNORE state 62
 - read/write printer ignore character 62
 - specify output streams 54
 - write printer driver type 60
 - write printer ignore character 62
 - write RS423 transmit rate 56
- output buffer event 38
- output streams 54

- Packed Decimal (P) 575
- palette 67
- panes 446
- pathname conventions 231
- pathnames 210
- pixel data, converting to screen 505
- pixel files format 514
- plot a sprite 422
- podule system manager 593
- pointer shape, changing 446
- pop-up menus 444
- printer ignore character 61
- printer stream 57

- procedure-call standard 623
- program environment 333

- RdchV 23
- ReadlineV 25
- redrawing windows 441
- registers 601
- registers, names for referring to 625
- RemV 25
- RS423 characters, interpretation of 175
- RS423 error event 39
- RS423 OS_Byte calls
 - read / write asynchronous communications state 177
 - read RS423 baud rates 180
 - read RS423 control byte 178
 - read/write RS423 busy flag 179
 - read/write RS423 ignore flag 180
 - read/write RS423 input buffer minimum space 180
 - read/write RS423 input interpretation status 178
 - write RS423 receive rate 176
- RS423 output stream 55
- RS423 port 174

- sample rate 528
- SCCB 544
- screen memory 321
- screen RAM 67
- single-character prefixes 186
- software vectors 16
- Sound Voice Generators 520, 522, 550
- sound channel control block 544
- sound channel handler 535
- sound channel interface 521
- sound DMA buffer handler 521, 523

- sound event 41
- sound event scheduler 522
- sound Level 0 * commands 524
- sound Level 0 SWI calls
 - configure the sound system 525
 - loudspeaker control 526
 - set stereo image position 527
 - sound system control 526
- sound Level 1 SWI calls
 - attach a channel to a named voice generator 539
 - attach channel to voice generator 539
 - convert pitch to internal representation 541
 - foreground (immediate) control of channel 541, 542
 - install voice generator 538
 - internal audio logarithm scaling 541
 - linear to audio logarithm 540
 - read channel control data 543
 - remove voice generator 538
 - set the overall loudness 540
 - set the sound system tuning 541
 - write channel control data 543
- sound Level 2 SWI calls
 - check free slots 547
 - flush and initialise the event queue 546
 - reserved Level 2 call 547, 548
 - schedule a sound event 546
 - set the sound system tempo 548
 - set/read the tempo beat counter 548
- sound Level 2 * commands 535, 545
- sound system 519
- sound system scheduler 545
- Sound_AttachNamedVoice 539
- Sound_Configure 525
- Sound_Control 541
- Sound_ControlPacked 542
- Sound_Enable 526
- Sound_InstallVoice 538
- Sound_LogScale 541
- Sound_Pitch 541
- Sound_QBeat 548
- Sound_QDispatch 548
- Sound_QFree 547
- Sound_QInit 546
- Sound_QRemove 547
- Sound_QSchedule 546
- Sound_QTempo 548
- Sound_ReadControlBlock 543
- Sound_RemoveVoice 538
- Sound_SoundLog 540
- Sound_Stereo 527
- Sound_Tuning 541
- Sound_Volume 540
- Sound_WriteControlBlock 543
- SoundChannels (Level 1) 521
- SoundChannels Level 1 control block 543
- SoundDMA 521
- SoundScheduler (Level 2) 522
- sprite * commands 418
- sprite area format 437
- sprite VDU commands 421
- sprites 417
- SpriteUtils module 417
- stack space, checking 184
- string scanning calls
 - General String Translation 414
 - string input 412
 - string input initialisation 411
 - substitute command line arguments 414

- string scanning routines 411
- SWI calls
 - ADFS 272
 - environment 335
 - mouse/pointer 134
 - sprite 422
 - debugger 572
 - Econet 585
 - heap manager 326
 - index of 641
 - modules 358
 - OS filing system 231
 - odule 593
 - sound Level 0 525
 - sound Level 1 538
 - sound Level 2 546
 - VDU 121
 - window manager 448
- SWI chunk number 385
- SWI decode code 388
- SWI decode table 388
- SWI error handling 8
- SWI handler code 385
- SWI instruction 4, 611
- SWI number conversions 409
- SWI numbers 6
- SWI Sound_AttachVoice 539
- SWI Sound_Speaker 526
- SWIV 27

- template files 447
- text and graphics 66
- text handling 443
- TickerV 29
- time 391

- time and date OS_Byte calls
 - 50Hz counter 396
 - read/write timer switch state 393
- time and date OS_Word calls
 - read CMOS clock 396
 - read interval timer 392
 - read system clock 391
 - write CMOS clock 398
 - write interval timer 393
 - write system clock 392
- tool windows 446
- transfer function, setting 508
- transient programs 352
- two-key rollover 147

- UkPlotV 27
- UKVDU23V 27
- UpCallV 29
- updating windows 442
- user event 40
- user flag, read/write 48
- user root directory 265

- VDU codes
 - back space 73
 - bell 73
 - carriage return 74
 - change display mode 82
 - clear graphics window 75
 - define graphics window 98
 - define text window 102
 - delete 103
 - disable printer 71
 - disable screen display 81
 - enable printer 71
 - enable screen output 72
 - form feed/clear screen 74

- general PLOT command 99
- home text cursor 103
- horizontal tab 73
- join cursors 72
- line feed 73
- miscellaneous commands 85
- next character to printer only 70
- no operation 102
- null operation 70
- page mode off 75
- page mode on 74
- position text cursor 103
- restore default colours 81
- restore default windows 102
- set graphics colour and action 76
- set graphics origin 102
- set palette 78
- set text colour 75
- split cursors 71
- vertical tab 74
- VDU control sequences 70
- VDU drivers 65
- VDU extension vector 134
- VDU OS_Byte calls
 - read / write general graphics information 112
 - read character at text cursor position and screen mode 110
 - read display screen bank number 108
 - read duration of first colour 104
 - read duration of second colour 105
 - read output cursor position 110
 - read text cursor position 110
 - read VDU driver screen bank number 107
 - read VDU status 109
 - read VDU variable value 111
 - read/write bell channel 114
 - read/write bell duration 115
 - read/write bell frequency 114
 - read/write bell sound volume 114
 - read/write bytes in VDU queue 115
 - read/write flash counter 113
 - read/write paged mode line count 115
 - reset font definitions 106
 - reset group of font definitions 106
 - set vertical screen shift and interlace 111
 - wait for vertical sync (vsync) 105
 - write display hardware screen bank 108
 - write duration of first colour 104
 - write duration of second colour 104
 - write shadow/non-shadow state 108
 - write VDU driver screen bank 107
- VDU OS_Word calls
 - read a character definition 117
 - read current and previous graphics cursor positions 119
 - read pixel logical colour 116
 - read the palette 118
 - write screen base address 119
 - write the palette 118
- VDU sequences 507
- VDU stream 56
- VDUXV 28
- vector code, writing 20
- vector SWI calls 17
- vectors 21
- vectors and handlers 351
- vectors, hardware 30
- voice generator header block 551
- VSYNC timers 395
- Vsync event 39

- WIMP environment 439
- WIMP error messages 486
- Wimp_CloseDown 485
- Wimp_CloseTemplate 483
- Wimp_CloseWindow 455
- Wimp_CreateIcon 451
- Wimp_CreateMenu 479
- Wimp_CreateWindow 449
- Wimp_DecodeMenu 480
- Wimp_DeleteIcon 454
- Wimp_DeleteWindow 454
- Wimp_DragBox 475
- Wimp_ForceRedraw 476
- Wimp_GetCaretPosition 478
- Wimp_GetIconInfo 472
- Wimp_GetPointerInfo 474
- Wimp_GetRectangle 468
- Wimp_GetWindowInfo 470
- Wimp_GetWindowState 469
- Wimp_Initialise 448
- Wimp_LoadTemplate 484
- Wimp_OpenTemplate 483
- Wimp_OpenWindow 455
- Wimp_Poll 456
- Wimp_ProcessKey 485
- Wimp_RedrawWindow 466
- Wimp_SetCaretPosition 477
- Wimp_SetExtent 482
- Wimp_SetIconState 471
- Wimp_SetPointerShape 482
- Wimp_UpdateWindow 467
- Wimp_WhichIcon 481
- window manager 439
- WordV 24
- WrchV 23
- | in VDU sequence 70
- *ACCESS 217
- *ADFS 265
- *APPEND 218
- *AUDIO 524
- *BACK 267
- *BACKUP 267
- *BREAKCLR 567
- *BREAKLIST 568
- *BREAKSET 568
- *BUILD 218
- *BYE 267, 283
- *CAT 218
- *CDIR 219
- *CHANNELVOICE 536
- *CLOSE 219
- *COMPACT 268
- *CONFIGURE 189
- *CONFIGURE FS 285
- *CONFIGURE LIB 286
- *CONFIGURE SoundDefault 537
- *CONFIGURE SpriteSize 421
- *CONFIGURE PS 291
- *CONTINUE 568
- *COPY 219
- *CREATE 221
- *DEBUG 569
- *DELETE 221
- *DIR 222
- *DISMOUNT 268
- *DRIVE 269
- *DUMP 222
- *ECHO 198
- *ENUMDIR 223
- *ERROR 198
- *EVAL 199
- *EX 223
- *EXEC 223

*EXEC file OS_Byte call 182
*EXEC file OS_Byte calls
 read/write *EXEC file handle 182
*EXEC input stream 181
*FORMAT 269
*FREE 269
*FREE 284
*FS 284
*FX 15, 199
*GO 199
*GOS 200
*HELP 200
*IF 201
*IGNORE 201
*INFO 224
*INITSTORE 569
*KEY 201
*LCAT 224
*LEX 224
*LIB 224
*LIST 225
*LOAD 225
*LOGON 284
*MAP 270
*MEMORY 569
*MEMORYA 570
*MEMORYI 571
*MODULES 356
*MOUNT 270, 285
*NAMEDISC 271
*NAMEDISK 271
*NODIR 271
*NOLIB 271
*NOURD 271
*OPT 225
*PASS 285
*PRINT 226
*PS 291
*QSOUND 546
*QUIT 571
*REMOVE 227
*RENAME 227
*RMCLEAR 356
*RMKILL 356
*RMLoad 357
*RMREINIT 357
*RMRUN 357
*RMTIDY 357
*RUN 227
*SAVE 227
*SCHOOSE 418
*SCOPY 418
*SCREENLOAD 418
*SCREENSAVE 419
*SDELETE 419
*SET 202
*SETEVAL 204
*SETMACRO 204
*SETPS 291
*SETTYPE 228
*SFLIPX 419
*SFLIPY 419
*SGET 419
*SHADOW 204
*SHOW 205
*SHOWREGS 571
*SHUT 228
*SHUTDOWN 228
*SINFO 420
*SLIST 420
*SLOAD 420
*SMERGE 420
*SNEW 420
*SOUND 537

- *SPEAKER 524
- *SPOOL 229
- *SPOOL stream 63
- *SPOOLON 229
- *SRENAME 421
- *SSAVE 421
- *STAMP 229
- *STATUS 205
- *STATUS FS 286
- *STATUS LIB 286
- *STATUS PS 291
- *STEREO 524
- *TEMPO 545
- *TIME 206
- *TITLE 271
- *TUNING 537
- *TV 206
- *TYPE 229
- *UNPLUG 358
- *UNSET 206
- *UP 230
- *URD 272
- *VERIFY 272
- *VOICES 536
- *VOLUME 536
- *WIPE 230

