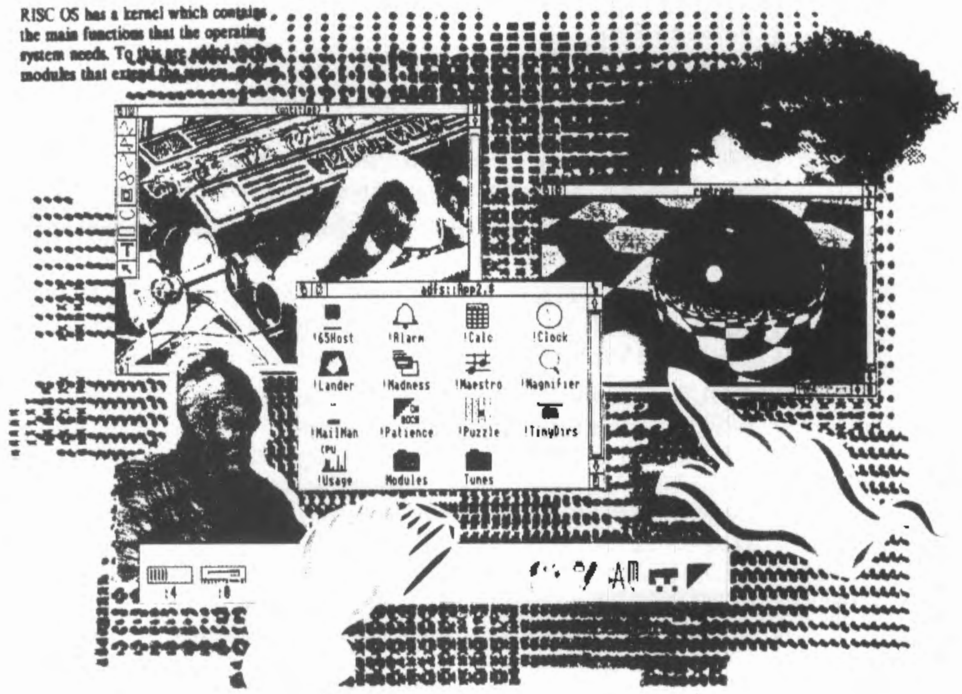
RISC OS
PROGRAMMER'S REFERENCE MANUAL
Volume I

Acorn
The choice of experience.

RISC OS
PROGRAMMER'S REFERENCE MANUAL
Volume I


RISC OS has a kernel which contains the main functions that the operating system needs. To this are added modules that extend the
adfs::App2.0
!65Host
!Alarm
!Calc
!Clock
!Lander
!Madness
!Maestro
!Magnifier
!MailMan
!Patience
!Puzzle
!TinyDirs
!Usage
Modules
Tunes

Acorn
The choice of experience.

All correspondence should be addressed to:

Customer Service
Acorn Computers Limited
Fulbourn Road
Cambridge CB1 4JN

Information can also be obtained from the Acorn Support Information Database (SID). This is a direct dial viewdata system available to registered SID users. Initially, access SID on Cambridge (0223) 243642: this will allow you to inspect the system and use a response frame for registration.

Within this publication, the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

Edition 1
Published 1989: Issue 1
ISBN 1 85250 060 3
Published by Acorn Computers Limited
Part number 0483,020

Contents

About this manual

Summary of contents

This manual gives you detailed information on the RISC OS operating system, so that you can write programs to run on Acorn computers that use it.

Part1

Part 1 introduces you to the hardware used to run RISC OS, and to the fundamental concepts of how RISC OS works.

Parts 2 to 5

Parts 2 to 5 inclusive give you more detailed information on separate parts of RISC OS:

- Part 2 describes the kernel (or central core) of RISC OS
- Part 3 describes the filing systems
- Part 4 describes the window manager
- Part 5 describes the system extensions to RISC OS

The information in these parts has been laid out as consistently as possible, to help you find what you need. Each chapter covers a specific topic, and in general includes:

- an Introduction, so you can tell if the chapter covers the topic you are looking for
- an Overview, to give you a broad picture of the topic and help you to learn it for the first time
- Technical Details, to use for reference once you have read the Overview
- SWI calls, described in detail for reference
- * Commands, described in detail for reference
- Application notes, to help you write programs
- Example programs, to illustrate the points made in the chapter, and for you to base your own programs on.

Appendices

The Appendices contain:

- an introduction to writing assembler for the ARM chip, on which RISC OS runs
- information of interest to RISC OS programmers writing compilers and other language-based tools
- file formats used by current RISC OS applications.

Tables

The tables gather together information from the whole manual, giving lists that you will find useful for quick reference.

Indices

The separate volume of indices contains:

- an index of * Commands
- an index of OS_Byte calls
- an index of OS_Word calls
- a numeric index of SWI calls
- an alphabetic index of SWI calls
- a main index.

Conventions used

Certain conventions are used in this manual:

Hexadecimal numbers

Hexadecimal numbers are extensively used. They are always preceded by an ampersand. They are often followed by the decimal equivalent which is given inside brackets:

&FFFF (65535)

This represents FFFF in hexadecimal, which is the same as 65535 in ordinary decimal numbers.

Typefaces

`Courier` type is used for the text of example programs and commands, and any extracts from the RISC OS source code. Since all characters are the same width in Courier, this makes it easier for you to tell where there should be spaces.

Command syntax

Special symbols are used when defining the syntax for commands:

- Angle brackets indicate that an actual value must be substituted. For example, <filename> means that an actual filename must be supplied.
- Braces indicates that the item enclosed is optional. For example, [K] shows that the letter 'K' may be omitted.
- A bar indicates an option. For example, 0|1 means that the value 0 or 1 must be supplied.

Programs

Many of the examples in this manual are not complete programs. In general:

- BBC BASIC examples omit any line numbering
- BBC BASIC Assembler programs do not show the structure needed to perform the assembly
- ARM Assembler programs assume that header files have been included that convert SWI names to SWI numbers (see the chapter entitled *An introduction to SWIs*)
- C programs assume that similar headers are included; they also do not show the inclusion of other headers, or the calling of *main()*.

Finding out more

For how to set up and maintain your computer, refer to the *Welcome Guide* supplied with your computer. The Welcome Guide also contains an introduction to the desktop which new users will find particularly helpful.

For details on the use of your computer and of its application suite, refer to the *User Guide* supplied with it.

If you wish to write BASIC programs on your RISC OS computer you will find the *BBC BASIC Guide* useful. Other specialist programming languages available from Acorn suppliers for RISC OS computers include:

- ANSI C
- Fortran 77
- ISO-Pascal
- Lisp
- Prolog

If you wish to write programs in assembly language, the *ARM Assembler* is available from your Acorn supplier.

Reader comments

If you have any comments on this Manual, please complete and return the form on the last page of the volume of Indices to the address given there.

Part 1 - Introduction

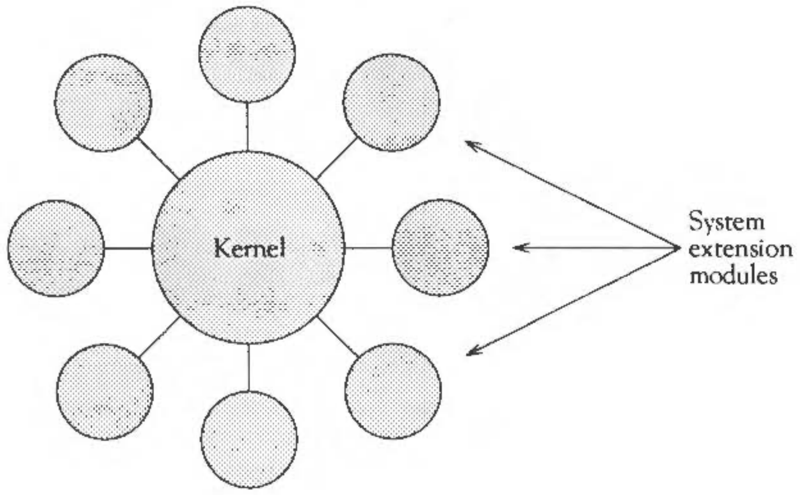
An introduction to RISC OS

Introduction

RISC OS is an operating system written by Acorn for its computers. Like any operating system, it is designed to provide the facilities that you, the programmer, need to control your computer and to get the most out of the programs you write for it.

Structure

RISC OS has a *kernel* which contains the main functions that the operating system needs. To this are added various *modules* that extend the system, adding such facilities as filing systems, a window manager, a font manager, and so on. These are called *system extension modules*:



The modules and the kernel provide their facilities very similarly, and there are few occasions when you will be able to distinguish whether the facilities you are using are provided by the kernel or by a system extension module. You are most likely to notice the difference if you wish to alter or replace part of the operating system.

Facilities

You can view RISC OS as a collection of routines that provide you with a wide range of facilities. You can get a good overview of the range that is covered from the earlier *Contents* pages of this manual.

This collection of routines can be broadly divided into three levels:

- those that RISC OS itself uses to automatically perform low-level tasks, such as *interrupt handling*
- those that provide sophisticated and powerful interfaces for you to use from programs, which are known as *Software Interrupts*, or *SWIs* for short
- those that provide simpler calls that can be used from the command line as well as from programs – these are the ** Commands* that you are probably already familiar with.

There are chapters later in this part of the manual that cover the above topics in more detail. They are entitled:

- *Interrupts and handling them*
- *An introduction to SWIs*
- ** Commands and the CLI*.

Altering and extending RISC OS

You can easily alter or extend RISC OS, because so much of it is written as modules.

Modules

Each of these modules conforms to a standard, which means that the facilities provided by the module are integrated into the system as if they were 'built-in'. You too can write modules that conform to this standard, so you can add things to RISC OS as you please.

You can also rewrite any of the standard RISC OS modules. Your replacement must provide the same entry points, and return values in the same way – but its internal workings can be functionally different. See the chapter entitled *Modules* for further details.

Vectors

Because the kernel is so large, it would not be easy for you to change it in the same way. You can instead make changes by using *vectors*.

A vector is a chain of entries that RISC OS uses to decide where to pass control to so it can perform a given function. Most vectors are used by SWIs. You can *claim* a vector, and redirect those SWIs to code of your own. Your code must accept the same input and provide similar output to the original SWI, but it can behave in a totally different manner – just like if you are replacing a module.

Some vectors are used by just one SWI, but others are used by several SWIs that perform similar functions. You can change how a whole group of SWIs behave by just claiming one vector – for example, SWIs that output characters.

A few vectors are not used by SWIs at all, but instead by other parts of RISC OS, to perform functions for which SWIs do not provide an interface.

See the chapter entitled *Software vectors* for more information.

How RISC OS is written

RISC OS is written in ARM assembler. This gives it some important advantages compared to writing it in a high level language such as C:

- the code produced is much more compact
- the operating system is faster, because there is less code involved in each task.

Of course, RISC OS can only be used on ARM-based computers.

To use RISC OS effectively, you should have a working knowledge of ARM assembler yourself. The chapter entitled *ARM Hardware* provides a brief introduction to the ARM processor and the set of chips that support it. The appendix entitled *ARM assembler* will give you a more detailed introduction to the ARM's assembly language.

How RISC OS is supplied

Because RISC OS is relatively compact, it is cost-effective to supply it in ROM chips. This also has advantages:

- it is much faster to start, as it does not need to be loaded into memory
- it cannot be easily lost or damaged, unlike disc-based operating systems.

There is an attendant disadvantage:

- it is harder to upgrade ROMs than a disc.

In practice, upgrades are done by patches that claim vectors or replace modules, as outlined above.

RISC OS and Arthur

RISC OS was developed from the Arthur operating system, which was the original operating system written for the Archimedes computer.

RISC OS is designed to be as compatible as possible with Arthur. Consequently, it supports some features of Arthur which have now been superseded. One example is the interrupt handling system, which has been much improved under RISC OS. However, old-style interrupt handlers written to run under Arthur will still work.

Arthur documentation

You will find that some minor parts of the Arthur operating system, which were in the old *Programmers Reference Manual*, are not in this manual. Instead, we've documented the new and preferred way of doing things. In general, we've only missed out the bits of RISC OS which have been included solely for compatibility with the Arthur operating system.

Some more major parts of the Arthur operating system are only referred to in passing. Again this is because they've been superseded.

In general though, RISC OS has extended most of the features of Arthur, as well as adding many of its own. So you will find most of the facilities of Arthur described in here, as well as a lot of new facilities.

Your old manuals

If you need full details of the old ways of doing things, so you can maintain old programs, you'll have to refer to your old manuals. So don't throw them away – keep them!

ARM Hardware

Introduction

To get the most out of your RISC OS computer, some knowledge of the hardware is important. This chapter introduces you to those features that are common to all RISC OS computers.

ARM chip set

Each RISC OS computer has a set of four chips in it, all designed by Acorn Computers Limited:

- an ARM (*Acorn RISC Machine*) processor, which does the main processing of the computer
- a VIDC (*Video Controller*) chip, which provides the video and sound outputs of the computer
- an IOC (*Input Output Controller*) chip, which provides the facilities to manage interrupts and peripherals within the computer
- a MEMC (*Memory Controller*) chip, which acts as the interface between the ARM, the VIDC chip, Input/Output controllers (including the IOC chip), and the computer's memory.

Together these chips are known as the *ARM chip set*.

Other components

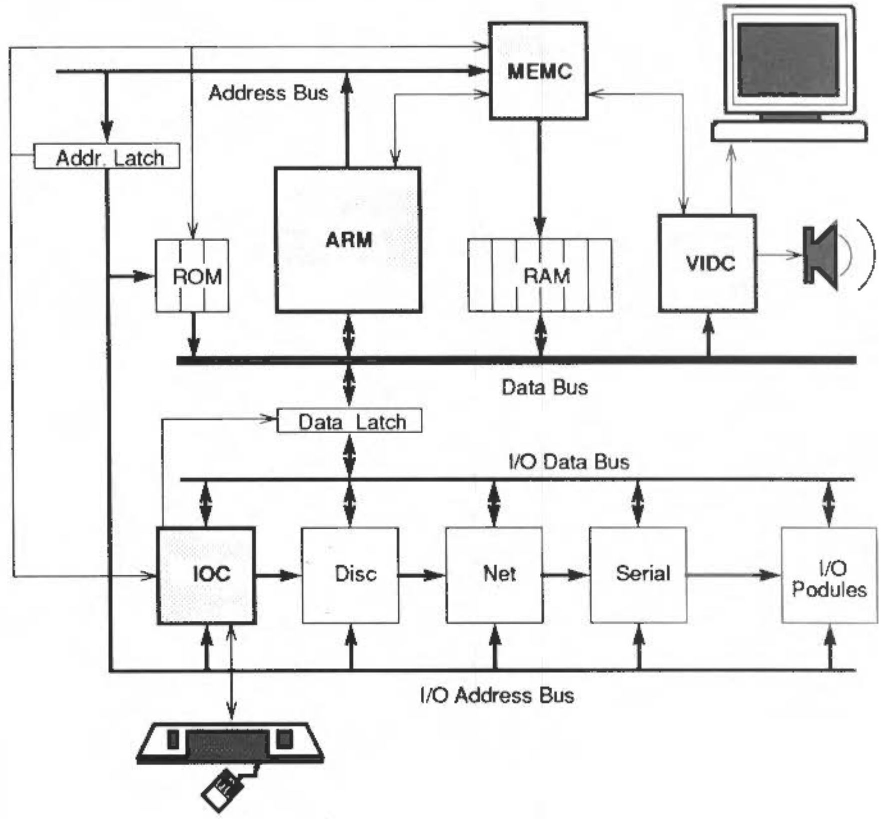
The other main electronic components of a RISC OS computer are:

- ROM (*Read Only Memory*) chips containing the operating system
- RAM (*Random Access Memory*) chips
- Peripheral controllers (for devices such as discs, the serial port, networks and so on).

Exactly which components and devices are present will depend on the model of computer that you have; see the Guides supplied with your computer for further details.

Schematic

The diagram below gives a schematic of an Archimedes computer, which may be viewed as typical of a RISC OS computer:



The ARM processor

The ARM is a RISC (Reduced Instruction Set Computer) processor – it has a comparatively small set of instructions. This simplicity of design means that the instructions can be made to execute very quickly.

RISC and CISC processors

A traditional CISC (Complex Instruction Set Computer) processor, as is commonly used as the main processor of a computer, provides a much larger and more powerful range of instructions, but executes them more slowly.

A CISC processor typically spends most of the time executing a small and simple subset of the available instructions. The ARM's instruction set closely matches this most commonly used subset of instructions. Thus, for the majority of the time the performance of the ARM is higher than that of comparable CISC chips; it is executing similar instructions more quickly.

The more complex instructions of a CISC chip are generally only occasionally used. For the ARM to perform the same task, several instructions may be necessary. Even then, the ARM still has a comparable performance, as it is replacing a single slow instruction by several fast instructions.

Summary

In summary, the simple RISC design of the ARM has these advantages:

- it has a high performance (four to five million instructions per second, or MIPS)
- it is cheaper to produce than CISC processors, making RISC OS computers cheaper for you to buy
- it is much simpler to learn how to program the chip effectively.

Word size

- The ARM uses 32 bit words. Each instruction fits in a single word. At any one time, the processor is dealing with three instructions:
- one instruction is executed
- the next instruction is simultaneously decoded
- the one after that is fetched from memory.

The ARM has a 32 bit data bus, so that complete instructions can be fetched in a single step. Its address bus is 26 bits wide, so it can address up to 64 Mbytes of memory (16 Mwords).

Processor modes

The ARM has four different modes it can operate in:

- User Mode, the mode normally used by applications
- Supervisor Mode (*SVC Mode*) used mainly by SWI instructions
- Interrupt Mode (*IRQ Mode*) used to handle peripherals when they issue interrupt requests
- Fast Interrupt Mode (*FIQ Mode*) used to handle peripherals that issue fast interrupt requests to show that they need prompt attention.

The last three modes are privileged ones that allow extra control over the computer. They have been used extensively in writing RISC OS.

Changing mode

Note that if you force the ARM to change mode (usually done using a variant of the `TEQP` instruction) you must follow this with a no-op (usually done using `MOVNV R0, R0`). This is to *avoid contention*, giving the ARM time to finish writing to the registers for one mode before switching to the other mode.

Registers

The ARM contains twenty-seven 32 bit registers; you can access sixteen of these in each of the modes. Some of the registers are shared across different modes, whilst others are dedicated to one mode. In the diagram below, registers dedicated to a privileged mode have been shaded grey:

User Mode	SVC Mode	IRQ Mode	FIQ Mode
R0			
R1... R6			
R7			
R8			R8_fiq
R9			R9_fiq
R10			R10_fiq
R11			R11_fiq
R12			R12_fiq
R13	R13_svc	R13_irq	R13_fiq
R14	R14_svc	R14_irq	R14_fiq
R15 (PC/PSR)			

Only two of the registers have special functions:

- R15 is used as the program counter (*PC*) and program status register (*PSR*)
- R14 (and R14_svc, R14_irq) are used as subroutine link registers.

One other set of registers is conventionally used by RISC OS for a special purpose:

- R13 (and R13_svc, R13_irq) are used as private stack pointers for the different processor modes.

All the remaining registers are general purpose.

R15 – program counter and status register

R15 contains 24 bits of program counter and 8 bits of processor status register:

31	30	29	28	27	26	25… …2	1	0
N	Z	C	V	I	F	Program counter (PC)	M1	M0

- bits 0 and 1 are the processor mode flags M0 and M1

00	User mode
01	FIQ mode
10	IRQ mode
11	SVC mode

- bits 2 - 25 are the program counter
- bit 26 is the FIQ disable flag F

0	Enable
1	Disable

- bit 27 is the IRQ disable flag *I*

0	Enable
1	Disable

- bits 28 - 31 are condition flags:

V	oVerflow flag
C	Carry flag
Z	Zero flag
N	Negative flag

R14 – subroutine link registers

R14 is used as the subroutine link register, and receives a copy of the return PC and PSR when a Branch and Link instruction is executed. It may be treated as a general purpose register at all other times. Similarly, R14_svc, R14_irq and R14_fiq are used to hold the return values of R15 when interrupts and exceptions arise, when Branch and Link instructions are executed within supervisor or interrupt routines, or when a SWI instruction is used.

R13 – private stack pointers

R13 (and R13_svc, R13_irq and R13_fiq) are conventionally used by RISC OS as private stack pointers for each of the processor modes.

If you write routines that are called from User mode and that run in SVC or IRQ mode, you will need to use some of the shared registers R0 to R12. You will therefore need to preserve the User mode contents on a stack before you alter the registers, and restore them before returning from your routine.

Note that the SVC and IRQ mode stacks must be full descending stacks, ending at a megabyte boundary. You are strongly advised not to change the system stack locations; if you do have to, you must be aware that they are reset to their default positions when errors are generated, and when applications are started.

FIQ routines need a faster response, so there are seven private registers in FIQ mode. In most cases these will be enough for you not to need to use any of the shared registers, and so you will be spared the overheads of saving them to a stack. If you do need to do so, you should for consistency use R13_fiq as the stack pointer.

You can use R13 and/or R13_fiq as conventional registers if you do not need to use them as stack pointers.

Instruction set

You will find details of the ARM's instruction set in *Appendix A - ARM Assembler*.

The VIDC chip

The VIDC chip controls and provides the computer's video and sound outputs. The data to control these systems is read from RAM into buffers in the chip, processed, and converted to the necessary analogue signals to drive the display's CRT guns and the sound system's amplifier.

The VIDC chip can be programmed to provide a wide range of different display formats. RISC OS uses this to give you 27 different screen modes. Likewise, you can program the way the sound system works.

Buffers

The VIDC chip has three buffers for its input data. These are used for:

- video data
- cursor data
- sound data.

Each of these buffers is first-in first-out (*FIFO*). The VIDC chip requests data from RAM as it is required, using blocks of four 32-bit words at a time. The MEMC chip controls the addressing and fetching of the data under direct memory access (*DMA*) control.

Video

Data from the video buffer is serialised by the VIDC chip into 1, 2, 4 or 8 bits per pixel. The data then passes through a colour look-up palette. The output from the palette is passed on to three 4-bit digital to analogue converters (*DACs*), which provide the analogue signals needed to drive the red, green and blue cathode ray tube (*CRT*) guns in the display monitor.

The palette has 16 registers, each of which is 13 bits wide. This supports a choice from 4096 different colours or an external video source.

The registers that control the video system give a wide choice of display formats:

- the pixel rate can be selected as 8, 12, 16 or 24 MHz
- the horizontal timing can be controlled in units of 2 pixels
- the vertical timing can be controlled in units of a raster
- the screen border can be set to any of the 4096 possible colours

If needed, support is provided for:

- interlaced displays
- external synchronisation
- very high resolution monochrome modes (up to 96 MHz pixel rate).

Cursor

The cursor data controls a pointer that is up to 32 pixels wide, and any number of rasters high (although RISC OS restricts the cursor to a maximum of 32 rasters in height). The cursor can use any three of the 4096 possible colours to colour its pixels. Alternatively, pixels can be marked as transparent, so that cursors can be any shape you desire.

The cursor may be positioned anywhere on the screen.

Sound

The sound data consists of digital samples of sound. The VIDC chip can support up to eight separate channels of sound. It provides eight stereo image registers, so the stereo position of each channel can be independently set.

The VIDC chip reads data from the buffer at a programmable rate. The data is passed to an eight bit DAC, which uses the stereo image registers to convert the digital sample to a stereo analogue signal. This is then output to the computer's internal amplifier.

The IOC chip

The IOC chip provides the facilities to manage interrupts and peripherals within your RISC OS computer. It controls an 8 to 32 bit Input/Ouput (I/O) data bus to which on-board peripherals and any I/O expansions are connected. It also provides a set of internal functions that are accessed without any wait states, and a flexible control port.

Internal functions

The following internal functions are provided by the IOC chip:

- Four independent 16 bit programmable counters. Two are used as baud rate generators – one for the keyboard, the other for the serial port. Another (*Timer 0*) is used to generate system timing events. The last timer (*Timer 1*) is unused by RISC OS, and you can program it for your own purposes.
- Six programmable bidirectional control pins.
- A full-duplex, bidirectional serial keyboard interface.
- Interrupt mask, request and status registers for both normal and fast interrupts.

Peripheral control

The IOC is connected to the rest of the ARM chip set by the system bus. It provides all the buffer control required to interface this high speed bus to the slower I/O or expansion bus. The IOC supports:

- sixteen interrupt inputs (14 level sensitive, 2 edge-triggered)
- seven external peripheral select outputs
- four programmable peripheral timing cycles (slow, medium, fast and 2 MHz synchronous).

Both the IOC and peripherals are viewed as memory-mapped devices. Most peripherals are a byte wide, and word aligned. A single memory instruction (LDRB to read, or STRB to write) can be used to:

- access the IOC control registers, or to
- select both a peripheral and the timing cycle it requires, and access it.

The IOC can support a wide range of peripheral controllers, including slower, low-cost periperal controllers that require an interruptable I/O cycle.

The MEMC chip

The MEMC chip interfaces the rest of the ARM chip set to each other and to the computer's memory. It uses a single clock input to provide all the timing signals needed by the chip set.

Memory support

MEMC provides the control signals needed by the memory:

- timing and refresh control for dynamic RAM (*DRAM*)
- control signals for several access times of read-only memory (*ROM*) – 450ns, 325ns, and 250ns.

Up to 32 standard DRAMs can be driven, giving 4 Mbytes of real memory using 1 Mbit devices.

Fast page mode DRAM accesses are used to maximise memory bandwidth, so that slow memory does not slow the system down too much.

Memory mapping

RISC OS computers use MEMC to map the physical memory into 16 Mbytes of logical memory. The base of this logical memory is at 32 Mbytes. RISC OS does not address this space directly, though; instead it addresses another 32 Mbyte logical slot within the 64 Mbytes logical address space supported by the ARM's 26-bit address bus. Each page of this slot can be:

- unmapped
- mapped onto one page of the logical memory
- mapped onto many pages of the logical memory.

RISC OS can only read and write from pages that have a one-to-one mapping. One-to-many mapping is used to 'hide' pages of applications away when several applications are sharing the same address (&8000 upwards) under the Desktop.

The computer's physical memory is divided into physical pages. Likewise, the 32Mbytes of logical space is divided into logical pages of the same size. MEMC keeps track of which logical page corresponds to which physical page, mapping the 26 bit logical addresses from the ARM's address bus to physical addresses within the much smaller size of RAM.

Page size

MEMC has 128 pages to use for its memory mapping. Each page has its own descriptor entry held in content-addressable memory. This simple structure allows the translation (of logical address to physical address) to be performed quickly enough that it does not increase memory access time.

In general, all 128 pages are used to map the RAM. Note that this is not always the case; for example, the Archimedes 305 uses only 64 pages.

If MEMC does use all 128 pages (or any other constant number), then:

- as the size of the computer's physical memory increases, the size of each page increases – a larger amount of physical memory is being split into the same number of pages
- as the size of each page increases, the number of logical pages decreases – the same amount of logical memory (32 Mbytes) is being split into larger pages.

The table below shows this. The values are those used in Archimedes computers, and may be viewed as typical of RISC OS computers. They should not be relied on for programming though; future RISC OS computers may not use 128 pages, leading to anomalies such as those in the first row (the Archimedes 305) :

Physical RAM Size	Page Size	No. of Logical pages
0.5 Mbyte	8 Kbytes	4 K
1 Mbyte	8 Kbytes	4 K
2 Mbytes	16 Kbytes	2 K
4 Mbytes	32 Kbytes	1 K

If you need to find out a machine's page size and so on, use OS_ReadMemMapInfo (SWI &51).

Number of pages programmed

RISC OS always programs 256 pages, even if it actually uses fewer pages. This is so that:

- random hits in the unused pages don't happen
- a second MEMC chip can be slaved to the master MEMC chip, allowing future machines to support 8 Mbytes of real memory.

Protection levels

Three protection levels are provided:

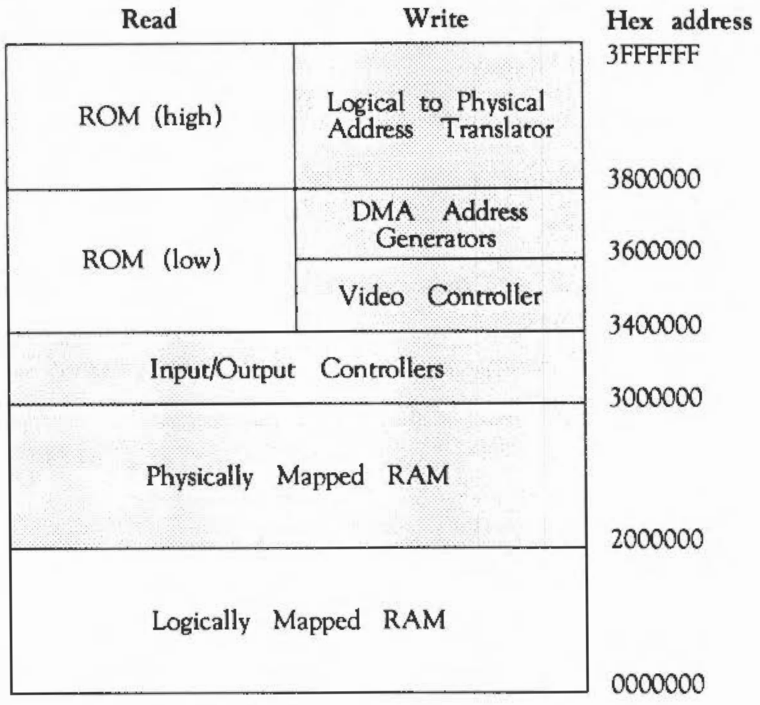
- Supervisor mode – this is the most privileged mode, that allows the whole address space to be freely accessed; it is available from all the ARM privileged modes (SVC, IRQ and FIQ)
- Operating System mode – this is more privileged than User mode when accessing logically mapped RAM, but acts as User mode in all other cases
- User mode – this is the least privileged mode, allowing access only to unprotected pages in the logically mapped RAM, and read cycles to the ROM space.

If an attempt is made to access protected memory from an insufficiently privileged mode, MEMC traps the exception and sends an abort signal to the ARM.

RISC OS does not use the Operating System mode.

Memory map

The resulting memory map is shown below. You can only access the areas shaded grey if you are in one of the ARM's privileged modes (SVC, IRQ or FIQ), which force MEMC to Supervisor mode by holding a pin high:



DMA support

MEMC also provides three programmable address generators to support direct memory access (DMA). They support:

- a circular buffer for video refresh
- a linear buffer for the cursor sprite
- double buffers for sound data.

Finding out more

If you need to find out more about ARM assembler and the ARM chip set, there are a number of sources you can turn to:

- ARM assembler is summarised in *Appendix A - ARM Assembler*
- ARM assembler is thoroughly covered in the manual supplied with the ARM *Assembler*, available from your Acorn supplier
- The ARM chip set is described in much greater detail in the *VL86C010 32-Bit RISC MPU and Peripheral User's Manual*, published by Prentice Hall.

In addition, a number of other publishers have produced books covering these topics – such is the interest in the ARM chip set.

An introduction to SWIs

Introduction

The main way you can access the routines provided by RISC OS is to use a SWI instruction. SWI stands for SoftWare Interrupt, and is one of the ARM's built-in instructions.

In brief, when you issue a SWI instruction, the ARM leaves your program. It jumps to a fixed location in memory, where there is a branch instruction into the RISC OS kernel code. This code examines the SWI instruction, and determines which particular OS routine you wanted. This is called, and when it is finished, control returns to your program.

The rest of the chapter will explain how to call SWIs from different languages, and follow how a SWI works in rather more detail.

SWI numbers and names

RISC OS can work out what routine you require because the SWI instruction code contains a 24-bit information field which identifies a routine uniquely. This field is known as the *SWI number*. A section later in this chapter describes how SWI numbers are allocated.

RISC OS provides several hundred different SWIs. You would find it difficult to remember what function each SWI number corresponds to, so each SWI also has a name. These names are held in the RISC OS ROMs, and in any system extension modules that have been loaded.

Parameters and results

Obviously, you need to be able to pass values to SWI routines (*parameters*), and must be able to read values back (*results*). The ARM registers are used to pass information between the user and RISC OS. In general, you will use R0 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, you generally store the data itself in the register. However, if the data is a string, or a large amount of numeric data, then you pass a pointer to the data instead. For example, filenames are passed as a pointer to the characters in memory, and the window manager uses pointers to large window descriptors.

An example

As an example of how to use a SWI, we will look at one called OS_WriteC. Its SWI number is &00. It is used to output a character. It takes a single parameter – the character you want to output – which is passed in R0. Suppose you wanted to output the character 'A', the ASCII code of which is 65.

Calling from Assembler

In assembler you could write:

```
        MOV    R0,#65        ; Load R0 with 'A'
        SWI    0             ; and output it
```

It would be clearer if you set a constant named `OS_WriteC` to &00. We suggest you do so in a standard header file that contains all SWI names and numbers. Using such a file, you could then write:

```
        MOV    R0,#65        ; Load R0 with 'A'
        SWI    OS_WriteC     ; and output it
```

When this is assembled, the bottom 24 bits of the SWI instruction are set to zero – the SWI number for OS_WriteC.

Calling from BBC BASIC

From BBC BASIC you can call a SWI routine in two different ways:

- use the built in assembler
- call it directly from BASIC.

BBC BASIC Assembler

BASIC's built in assembler is very similar to the standard ARM assembler. However, the SWI names are available as strings; note that this means you must enclose them in double quotes. The case of the letters is significant:

```
MOV    R0,#65        ; Load R0 with 'A'
SWI    "OS_WriteC" ; and output it
```

BBC BASIC

You can use the BASIC keyword `SYS` to call SWI routines directly from interpreted BASIC. BASIC just asks RISC OS what SWI number the given string corresponds to; you will find full details of the syntax in the *BBC BASIC Guide*. Our example would be written:

```
SYS "OS_WriteC",65
```

Calling from C

The Acorn C library provides a similar procedure to call a SWI routine. Again, you should see the *ANSI C* manual for full details of the syntax, and how errors are handled. The example below assumes that relevant header files have been `#included`:

```
_kernel_swi_regs regs;                    /* declare register structure */

regs.r[0] = 65;                           /* set pseudo R0 to 'A' */
_kernel_swi(OS_WriteC, &regs, &regs);     /* call SWI */
```

More about SWI numbers and names

In general, you don't have to worry about the exact mechanism used by RISC OS to decode the SWI instructions. As long as you use the right SWI number, and pass the correct parameters, the correct result will be obtained.

We strongly advise you to use SWI names in your code, for added clarity. This is easy from BASIC, as the names are already set up; from other languages (such as assembler and C above) you will find this easiest if you set up header files. **Examples in the rest of this manual will assume you have done so.**

SWI name prefixes

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 RISC OS. Examples of other prefixes are Font, Wimp, and ADFS. The prefix is determined by the module which implements the SWI.

Error handling – an introduction

RISC OS provides full error handling facilities for SWIs. In general, if a SWI has no errors, the V flag in R15 is clear as the routine exits; if there is an error, the V flag is set and R0 points to an error block on exit.

As the routine exits, RISC OS checks the V flag. If it is set (meaning there was an error), then RISC OS looks at bit 17 (the *X bit*) of the SWI number:

- if it is set then control returns to your program, and you should deal with the error yourself
- if it is clear control is passed to the system error handler, which reports the error to you. You can of course replace the system error handler with one of your own; indeed, most programs do.

For further details, see the chapter entitled *Generating and handling errors*.

SWI numbers in detail

The 24 bits used to encode the SWI number in the instruction allow SWIs in the range 0 - &FFFFFF (16777215) to be used. This SWI 'address range' is divided up into several parts under RISC OS. For example, SWIs in the range 0 - &3FFFF (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.

You can also define your own SWI calls. 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 return the error `No such SWI`. However, a user program can claim 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 that the SWI expects to be in the machine. All SWIs used under RISC OS have these bits set to zero. Under RISC iX, bit 23 is set to 1 and all other bits are set to zero.

Bits 18 - 19

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

Bit number 19	18	**Meaning**
0	0	Operating system
0	1	Operating system extension modules
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, device drivers for expansion cards, and the Font manager have bit 18 of their SWIs set, so their SWI numbers start at &40000. Note that this can include system extension modules written by third parties.

Any SWIs provided by application software that is 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 in the chapter entitled *Generating and handling errors*.

Bits 6 - 16

These are the SWI Chunk Identification numbers. They identify a block of 64 consecutive SWIs, for use within a single application or system extension module. Anyone wishing to use one of these blocks of SWIs for distributed software should apply in writing to Acorn Customer Service, who will allocate a unique value.

Bits 0 - 5

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

```
000010nnnnnnnnnnn000000 to
000010nnnnnnnnnnn111111
```

where nnnnnnnnnnn is the chunk number that the software house has been allocated for the application or module.

Technical details

Although in general you don't need to know how a SWI is decoded and executed, there are some more advanced cases where you will need to know more. This is what happens:

1 The contents of R15 are saved in R14_svc (the SVC mode subroutine link register).

2 The M0 and M1 bits of R15 are set (the processor is forced to SVC mode) and the I bit is also set (IRQ is disabled).

3 The PC bits of R15 are forced to &08.

4 The instruction at &08 is fetched and executed. It is a branch to the code that RISC OS uses to decode SWIs.

5 RISC OS uses the PC bits of the return address held in R14_svc to pick up a copy of the SWI instruction.

6 Interrupts are restored to the state they were in when the SWI was issued. This is done by setting the I bit in R15 to the value of the equivalent bit in R14_svc.

7 The V bit of the return address held in R14_svc is cleared, unless the SWI was OS_BreakPt or OS_CallAVector. (This is done for the error handling system – see the chapter entitled *Generating and handling errors*.)

8 RISC OS looks at the 24 bit SWI number field held in the SWI instruction, and uses it to decide where to branch to.

9 If the SWI does not use a vector, RISC OS will branch directly to the actual SWI routine.

If the SWI does use a vector, RISC OS branches to the routine that calls the vector. Unless you have claimed the vector, this will execute the actual SWI routine.

10 The SWI routine is executed.

11 Any error handling is performed.

12 Any call back handling is performed.

13 Control is returned to your program by using the instruction `MOVS R15, R14_svc`. This restores both the mode you were in when you called the SWI, and the interrupt status. Note however that a few SWIs (such as OS_IntOn, which enables interrupts) deliberately alter the mode and/or interrupt status so they are not restored on exit.

If an error is being returned by setting the V bit, the instruction `ORRS R15, R14_svc, #V_bit` is used instead.

An example of documentation

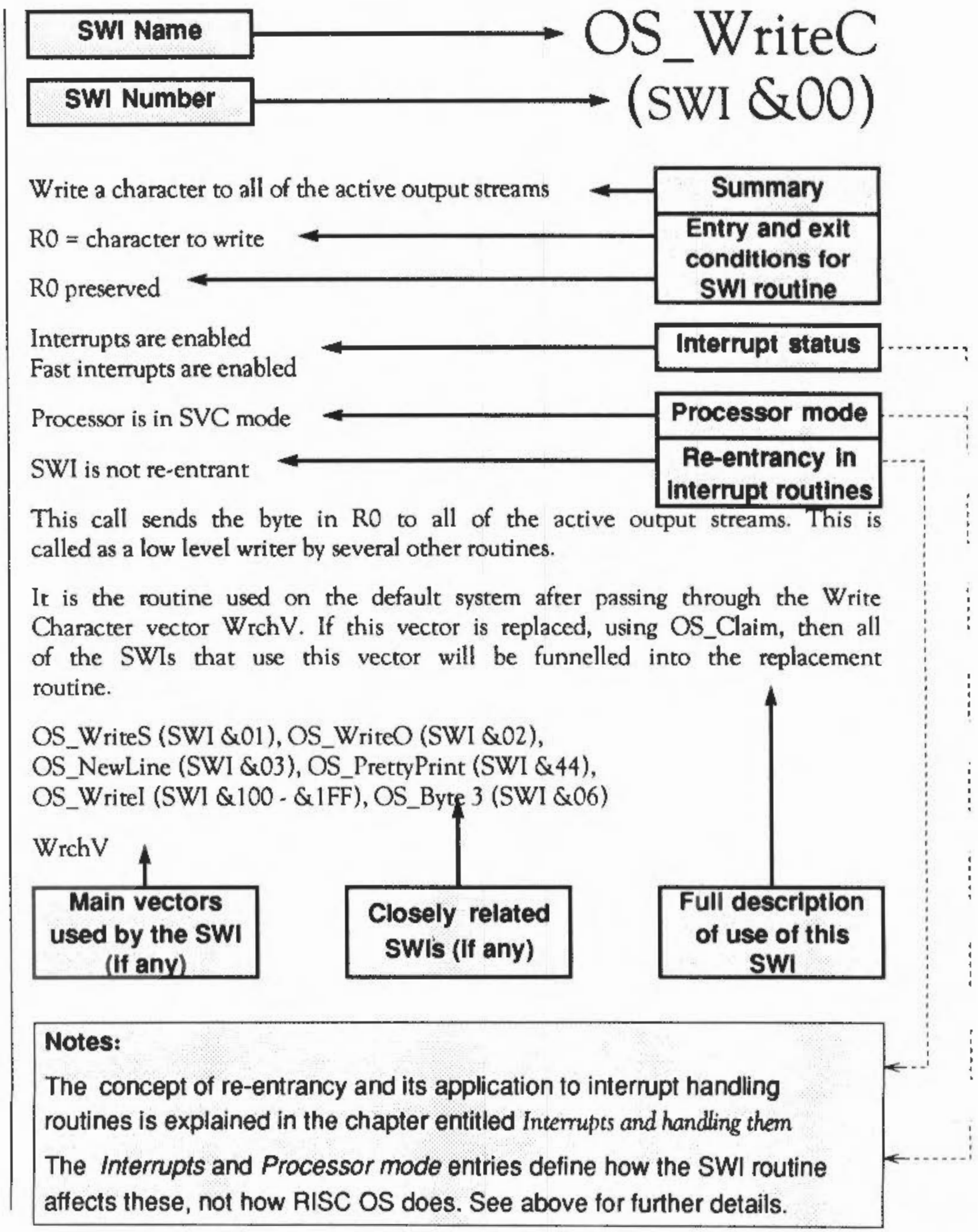
Below is an example of how a SWI is documented. Comments are provided in grey boxes so you can understand exactly what each bit means.

Some things are assumed to be consistent for all SWIs, and only exceptions are documented:

- SWIs are decoded and executed as outlined above
- The V flag is cleared if there is no error; it is set if there is an error, and R0 will then point to an error block. See the chapter entitled *Generating and handling errors* for further details

Other registers and flags are preserved across the call unless stated otherwise.

Note that the description of the SWI refers to the routine itself – in other words, what happens during step 10 above. Thus headings such as *Processor mode* and *Interrupts* refer to what happens in the SWI routine itself – not what happens when the SWI is decoded etc.


SWI Name
SWI Number
Summary
Entry and exit conditions for SWI routine
Interrupt status
Processor mode
Re-entrancy in interrupt routines
Main vectors used by the SWI (if any)
Closely related SWIs (if any)
Full description of use of this SWI

OS_WriteC
(SWI &00)

Write a character to all of the active output streams

On entry — R0 = character to write

On exit — R0 preserved

Interrupts — Interrupts are enabled
Fast interrupts are enabled

Processor mode — Processor is in SVC mode

Re-entrancy — SWI is not re-entrant

Use — This call sends the byte in R0 to all of the active output streams. This is called as a low level writer by several other routines.

It is the routine used on the default system after passing through the Write Character vector WrchV. If this vector is replaced, using OS_Claim, then all of the SWIs that use this vector will be funnelled into the replacement routine.

Related SWIs — OS_WriteS (SWI &01), OS_WriteO (SWI &02),
OS_NewLine (SWI &03), OS_PrettyPrint (SWI &44),
OS_WriteI (SWI &100 - &1FF), OS_Byte 3 (SWI &06)

Related vectors — WrchV

Notes:

The concept of re-entrancy and its application to interrupt handling routines is explained in the chapter entitled *Interrupts and handling them*

The *Interrupts* and *Processor mode* entries define how the SWI routine affects these, not how RISC OS does. See above for further details.

Important notes

There are some important points to note if you are writing your own SWI routines. These apply if:

- you call a SWI from your own SWI routine
- you claim a vector and replace a routine with one of your own.

Calling SWIs from SWI routines

Normally SWIs are executed in SVC mode. If you call another SWI from this mode without taking precautions, it will use R14_svc and crash the computer as follows:

1. The first SWI is executed from a program that is running in User mode. R15 (the return address to the program) is copied to R14_svc, and the processor is put into SVC mode.
2. The first SWI routine is entered.
3. This routine calls the second SWI from SVC mode. R15 is copied to R14_svc, overwriting the return address to the program. The processor remains in SVC mode.
4. The second SWI executes, and control is returned to the first SWI routine by loading R14_svc back into R15.
5. The first SWI routine finishes executing, and tries to return control to the program by loading R14_svc back into R15.
6. Because R14_svc was overwritten by the second SWI, control is not returned to the program. Instead, the computer just repeatedly loops through the end of the first SWI routine.

The cure is simple; you must save R14_svc before you call a SWI from within another SWI routine, and restore it after control returns to the SWI routine. This is typically done using a full descending stack pointed to by R13_svc, like this:

```
STMFD   R13!, {R14}             ; Save return address
SWI ...                         ; Call the SWI (corrupts R14)
LDMFD   R13!, {R14}             ; Restore return address
```

Of course if you call several SWIs, you don't have to save and restore R14 around each call – instead you should save it before calling the first SWI, and restore it after the last one.

Error handling with vectored SWIs

Normally, you can assume that the V flag of the return address held in R14_svc has been cleared by RISC OS before a SWI routine is entered. This leaves the return address in the correct format to indicate that no errors occurred.

You cannot make this assumption for SWI routines that are vectored. This is because any of these routines might be called using the SWI OS_CallAVector, for which RISC OS does not clear the V bit.

Therefore, if you claim a vector and replace a SWI routine with one of your own, that routine must not assume the state of the V flag. Instead, you must explicitly clear the V flag if there was no error, or explicitly set it (and set up an error block) if there was an error.

* Commands and the CLI

Introduction

* *Commands* provide you with a simple way to access the facilities of RISC OS by using text – for example:

```
*Time
```

will display the time and date. If you have read your computer's *User Guide*, you will already be familiar with many of these commands.

This chapter introduces you to * Commands and the CLI; a later chapter entitled *The CLI* describes them in more detail.

Command Line mode

Perhaps the most common way of issuing a * Command is to type it when the computer is in *Command Line mode* – also called *Supervisor mode* by some screen displays. Each line starts with a * character prompt, so you don't need to type it yourself. In the above example, all you need to type is the text `Time` .

OS_CLI and the CLI

When you type a * Command, the text is passed to RISC OS by a SWI, named OS_CLI. The text is then interpreted by a part of RISC OS called the *Command Line Interpreter* – or *CLI* for short. This converts the text to one or more SWIs that do the work of the * Command.

For example, the *Time command just calls three SWIs. You can achieve the same effect with a few lines of BASIC:

```
DIM block 24
?block = 0
SYS "OS_Word",14,block
SYS "OS_WriteN",block,24
SYS "OS_NewLine"
```

The * Command version is obviously more convenient.

* Commands v. SWIs

* Commands have a number of advantages when compared to SWIs, mainly because of their simplicity:

- they are simple for novice users to use
- they can be easily typed in directly, either from the command line or from applications
- they are simpler to call from programs
- they provide simple access to powerful features.

Their simplicity also leads to some disadvantages:

- they are not as flexible as SWIs
- they cannot easily pass information back to a program, as they usually output results to the screen.

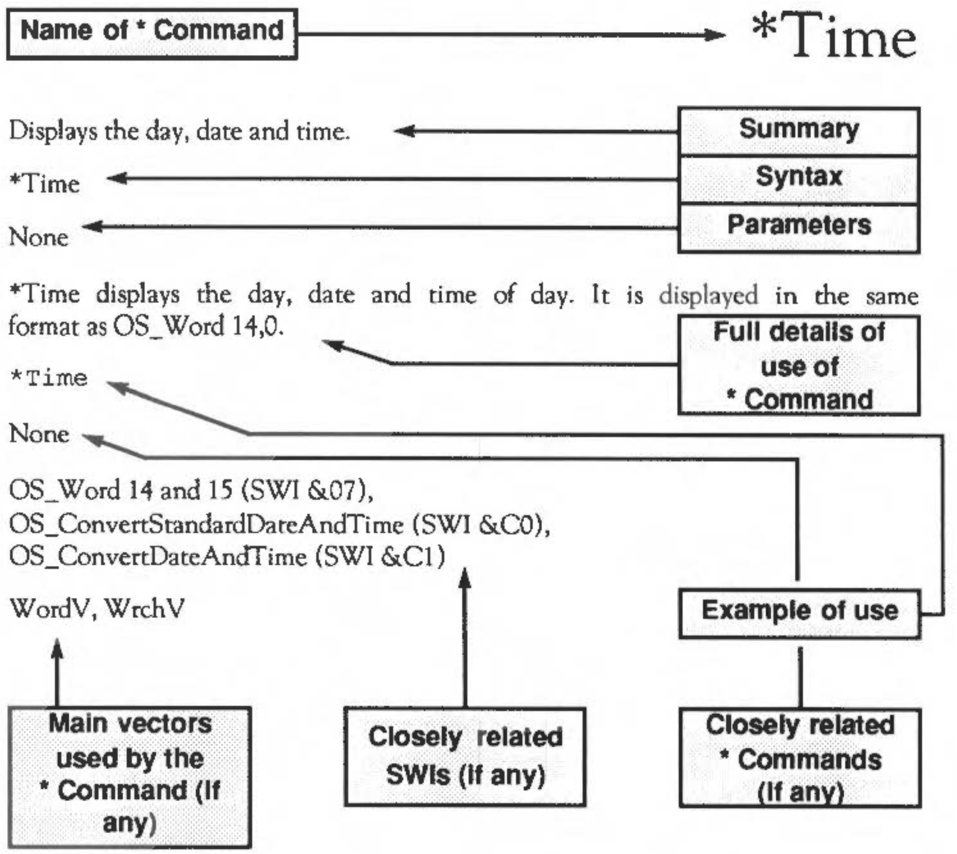
It is up to you whether you use * Commands or SWIs. Sometimes you will have to use SWIs, so you can do something that * Commands do not cater for. There will be other times when you use * Commands for their simplicity and ease of use.

Documentation

Each * Command is documented in the relevant chapter. For example, *Time is described in the chapter entitled *Time and date*. You will find many of the miscellaneous * Commands that the kernel supplies, in the chapter entitled *The* CLI. (This chapter also details the OS_CLI SWI.)

An example of documentation

The next page gives an example of how a * Command is documented. Again, comments are provided in grey boxes so you can understand exactly what each bit means:


Name of * Command
Summary
Syntax
Parameters
Full details of use of * Command
Example of use
Main vectors used by the * Command (if any)
Closely related SWIs (if any)
Closely related * Commands (if any)

*Time

Displays the day, date and time.

Syntax

*Time

Parameters

None

Use

*Time displays the day, date and time of day. It is displayed in the same format as OS_Word 14,0.

Example

```
*Time
```

Related commands

None

Related SWIs

OS_Word 14 and 15 (SWI &07),
OS_ConvertStandardDateAndTime (SWI &C0),
OS_ConvertDateAndTime (SWI &C1)

Related vectors

WordV, WrchV

Using * Commands

You don't have to be in Command Line mode to use * Commands. In fact, you can call * Commands in a number of other ways – both from applications and programming languages. The sections below outline these.

Issuing * Commands from applications

You can issue * Commands from most applications by typing a * at the start of a command. The application recognises the * prefix and calls OS_CLI, instead of trying to execute it itself.

When you write applications, they too should recognise any * prefixes, and call OS_CLI.

Issuing * Commands from assembler

You can issue * Commands from assembler by passing the string directly to the SWI OS_CLI. Note the terminating null byte:

```
                ADR R0,TIMESTR          ; Make R0 point to the text
                SWI OS_CLI              ; and call OS_CLI
                ...

TIMESTR         DCB "Time",0            ; Define the * Command text
                ALIGN
```

Issuing * Commands from BASIC

There are a number of different ways you can issue * Commands from BBC BASIC.

Directly from programs

You can issue them directly from your program:

```
*TIME
```

The OSCLI keyword

Sometimes you won't know all the text of the * Command you want to use; for instance, you might want the user of your program to give the name of a file. Instead of issuing the command directly, you can build up the text of the * Command, and then use the OSCLI keyword:

```
INPUT "Name of file to delete"; file$
OSCLI "Delete "+file$
```

Calling OS_CLI directly

Of course, you can also call OS_CLI directly, as outlined in the chapter *An introduction to SWIs*. You can either use the `SYS` keyword:

```
DIM TIMESTR 4
$TIMESTR = "TIME"
SYS "OS_CLI",TIMESTR
```

or more simply:

```
SYS "OS_CLI", "TIME"
```

or you can use BBC BASIC's built-in assembler:

```
              ADR R0,TIMESTR          ; Make R0 point to the text
              SWI "OS_CLI"            ; and call OS_CLI
              ...

.TIMESTR      EQUS "TIME"             ; Define the * Command text
              EQUS 0
```

See the *BBC BASIC Guide* for full details of all the above syntax.

Issuing * Commands from C

Similarly, the Acorn C library provides different ways for you to issue * Commands.

The procedure system ()

You can use the procedure `system`, which takes as a parameter the text of the * Command:

```
system("Time");
```

You cannot run a replacement application using this call, unless prefixed with "CHAIN: ". So:

```
system("BASIC");
```

would start up BBC BASIC, but when BASIC quits control returns to the C application rather than its parent.

Calling OS_CLI directly

Alternatively, you could directly call OS_CLI:

```
_kernel_swi_regs regs;
char timestr[] = "Time";

regs.r[0] = (int) timestr;
_kernel_swi(OS_CLI, &regs, &regs);
```

Changing and adding * Commands

One of the keynotes of RISC OS is the ease with which you can alter and extend it. You've already been introduced to how you can alter, replace or add SWIs. The techniques that can be used for this are:

- claiming vectors
- replacing modules
- adding new modules.

In just the same way, you can use these techniques to alter, replace or add * Commands.

Using vectors

If you claim a vector, and hence change how the SWI that uses it works, you will also alter all functions of RISC OS that call that SWI – including * Commands.

As an example, let's assume that you have changed OS_WriteC so that all letters are converted to capitals. You'd do this by claiming WrchV, the vector used for character output, so that it passes on calls made to OS_WriteC to your routine instead.

This would mean that all * Commands that output their results via WrchV would now do so in capitals only. This is true of all * Commands that output characters, and our example of `*Time` is no exception.

See the chapter entitled *Software vectors* for further details of how to use vectors.

Replacing modules

If you replace a module, you must provide the same services that the old module did. So your replacement module should have the same * Commands, each of which must have the same syntax and accept the same parameters as before. However, they can be functionally different.

There is no reason why a replacement module cannot add extra * Commands as well.

Adding modules

If you write a new module, it can provide * Commands, in exactly the same way as any of the system modules. See the chapter entitled *Modules*, for details of how to write a module.

Generating and handling errors

Introduction

It is reasonable to expect that most SWIs can generate an error. For example, if you pass poor parameters 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 routine will clear the ARM's V (overflow) flag on exit. If an error did occur, the SWI routine will set V on exit. Furthermore, R0 will contain a pointer to an error block, which is described below.

Error handling

Just before RISC OS passes control back to your program, it checks the V flag. If it is clear (no error occurred) control passes directly back.

If V is set (an error occurred), RISC OS looks at a copy of the original SWI instruction you used:

- If you had cleared bit 17 of the SWI number, RISC OS deals with the error itself. Control does not return normally to your program; instead the error is passed to the error handler used by your program, which normally will report the error to you.
- If you had set bit 17 of the SWI number, RISC OS returns control directly to your program. The V flag will still be set to indicate an error, and R0 will contain the error pointer. It is up to you to deal with the error.

Types of SWIs

These two types of SWI are known respectively as *error-generating* 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. All the examples in the chapter entitled **Commands and the CLI* were error-generating SWIs. If you want to call an error-returning SWI, with bit 17 set:

- add &20000 to the SWI number you use, or:
- put the letter **X** in front of the SWI name, thus: XOS_WriteC, XWimp_OpenWindow, and so on.

Error blocks

The error block pointed to by R0 has the following format:

R0 + 0	a word containing the error number
R0 + 4	error message, terminated by a zero byte.

An error block must be word-aligned, and must be no more 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.

The bottom byte is often a basic 'error number'.

The middle two bytes identify what generated the error. Third parties generating their own errors should apply to Acorn for an identifier. The following error ranges have 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	FileSwitch errors
&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 filing system number &XX
(eg &10800 - &108FF	ADFS errors)
&20000 - &200FF	Sound errors

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) or floating point exception, from which it wasn't possible to sensibly return with V set. In such cases different error ranges are used:

&80000000 - &800000FF	Machine exceptions
&80000100 - &800001FF	CoProcessor exceptions
&80000200 - &800002FF	Floating Point exceptions
&80000300 - &800003FF	Econet exceptions

- Bit 30 is defined to be clear, and can therefore be used by programmers to flag internal errors.
- Bits 24 - 29 are reserved. They should be cleared for compatibility with any future extensions.

  If bit 31 is set then these bits are sometimes used as a suberror indicator; for example ADFS uses them to show what kind of disc error has occurred.

Technical details of error-generating SWIs

You may need to know in more detail how RISC OS handles an error that an error-generating SWI creates.

1 First it informs modules of the error using the SWI OS_ServiceCall, with reason code 6 (Error). This is for the module's information only, so that it can tidy up (close files, and so on) before RISC OS handles the error. The module must not try to handle the error.

2 It then calls the error vector (see the later chapter entitled *Software vectors*). By default, this calls the current error handler. You may claim this vector, but again this should be for information only – for example, so that your program can tidy up. The call must subsequently be passed on to the error handler; your program must not try to handle the error.

If you want to handle an error yourself, you must instead use the error-returning version of the SWI.

Generating errors

In addition to detecting errors, you might want to generate an error which calls the current error handler, so you can find out about a problem. A common example would be if you detect that `Esc` is pressed. This is usually a sure sign that the user wants to abandon the current operation. The standard response is for you 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, you should call the SWI OS_GenerateError. On entry, R0 contains a standard error block pointer. The routine never returns. 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.

Writing system extension code

You must not write system extension code (such as a module, interrupt handler or transient) that generates errors – users of this code have a right to expect it to work. This means that you must always use the X form of SWIs in such code.

The only time you should call OS_GenerateError from system extension code is to report exception-type errors – that is, when bit 31 of the error number is set. For example, the Floating Point Emulator uses this mechanism to report exceptions from both the hardware and software floating point processors, as coprocessor instructions obviously cannot return with the V bit of the ARM processor set to indicate an error.

OS_GenerateError (SWI &2B)

Generates an error and invokes the error handler

On entry

R0 = pointer to error block

On exit

Doesn't return – OS_Generate Error (SWI &2B) or
V flag is set – XOS_Generate Error (SWI &2002B)

Interrupts

Interrupts are enabled by OS_GenerateError, but unaltered by the X form
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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
   ADRCS R0,escapeBlock                ; Get ptr. to error block
   SWICS "OS_GenerateError"            ; Do the error - doesn't
                                       ; return
.noEscape
   ....
.escapeBlock
   EQUD  17                            ; Error number for escape
   EQUS  "Escape"+CHR$0                ; Error string
   ALIGN
```

Related SWIs

None

Related vectors

ErrorV

*Error

Returns errors

Syntax	`*Error [<number>] <text>`
Parameters	`<number>` a number `<text>` a string of printable characters
Use	*Error returns an error with the specified error number and the explanatory text. This is normally then printed on the screen. Programmers will find this command useful for reporting errors after trapping them within a command script. If the error number is omitted it is set to the default value of 0.
Example	`*Error 100 No such file` will print 'No such file (Error number &64)'
Related commands	None
Related SWIs	OS_GenerateError (SWI &2B)
Related vectors	ErrorV

OS_Byte

Introduction

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 SWIs 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 older 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 manual. This chapter documents OS_Byte in general terms, so that when examples of its use are given later on, you will understand the entry and exit conditions better. The next chapter outlines how OS_Word works.

Parameters

OS_Byte takes one, two or three parameters. The first parameter, passed in R0, is the reason code. This indicates which particular action you require OS_Byte to take. 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 table entitled *OS_Bytes*, which you will find at the end of the manual.

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.

In fact, all OS_Byte routines mask out the top 24 bits of the parameters when they use them. Although these top bits are not used, calls to OS_Byte always preserve them in R0; the same applies for R1 and/or R2 where they are documented as preserved. If you are writing an OS_Byte routine (or one to decode them), you must make sure you preserve the top 24 bits, at least in R0. This means you will have to mask the parameter(s) into a temporary register, rather than back into the passed parameters.

Some OS_Byte calls return values. On earlier Acorn computers these were always byte-wide, but on RISC OS computers some of these values may now be too large to fit in a single byte, and should be treated as whole words. For example, if you were reading the number of spaces left in a buffer using OS_Byte 128, you might read the two 'byte' result returned in R1 (low byte) and R2 (high 'byte' – in fact 24 bits) like this:

```
ADD    Rn,R1,R2,LSL#8
```

Calling OS_Byte

You call the OS_Byte SWI in exactly the same way as any other SWI. See the chapter entitled *An introduction to SWIs* for details.

The calls may be grouped into three main classes, according to the value of R0 on entry.

Calls where R0 is less than 128

If R0 is less than 128, then only R1 is used to pass further information. However, R2 is often used as a temporary register and corrupted in the process. You use these calls to set *status variables*, which the computer uses to control its operation. For example, OS_Byte 5 sets the status variable for the type of printer that is connected.

In addition to setting the appropriate status variable, these calls may also perform some other task. For example, OS_Byte 5 also waits for the current printer buffer to become empty before returning. Although these calls sometimes return the 'previous' state of the status variable, they are normally used for the action they perform, rather than the information they return.

Calls where R0 is between 128 and 165

If R0 is between 128 and 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.

Calls where R0 is between 166 and 255

For calls with R0 between 166 and 255 on entry, the action is always the same. R0 acts as an index into the RAM which holds the status variables. They are held in consecutive memory locations, so R0=166 accesses the first one, R0=167 accesses the second one, and so on. The contents of R1 and R2 determine what happens to the status variable:

New Value = (Old Value AND R2) EOR R1

On exit, R1 holds the old value of the status variable, and R2 holds the value of the status variable in the next memory location.

Reading and writing values

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

Altering selected bits

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 status variable. You should:

- clear the bits of R2 corresponding to the bits you want to alter
- set the corresponding bits of R1 to the new value you want these bits to have.

For example, to set bits 2 - 4 of a status variable to the binary pattern 101, and leave the rest unaltered, you would use:

R2 = &E3 (11100011 in binary) and
R1 = &14 (00010100 in binary)

In all cases, the calls in the range 166 - 255 return with the previous value of the variable in R1 and the value of the next variable in RAM (ie the one which would be accessed with R0+1) in R2. The exception is where R0 = 255, where there is no defined 'next' location, and so the value of R2 is undefined.

Altering any of these variables does not have any immediate effect, but may often seem to, as many are acted upon by interrupt routines.

Which call to use when

Many of the calls in this last group access the same status variable as the low-numbered calls, between 0 and 127. 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.

OS_Byte and interrupts

Like most important SWIs, OS_Byte is vectored so you can alter how it works. Before its vector is called, interrupts are disabled. Most OS_Byte routines are so short that there is no need for them to re-enable interrupts – instead they rely on RISC OS doing this when control is returned to you. Because these OS_Byte routines do not re-enable interrupts they are also used by interrupt handling routines

If you replace or alter an OS_Byte routine, make sure that:

- you do not change the way it alters the interrupt status
- you do not make it take so long that interrupts are disabled for an unreasonably long time.

Adding OS_Byte calls

You can add your own OS_Byte calls to RISC OS by installing a routine on the software vector that OS_Byte calls use. For full details, see the chapter entitled *Software vectors*.

There is an alternative, but less preferable way of adding OS_Byte calls. If you issue an OS_Byte with a number that RISC OS doesn't recognise, it issues an Unknown OS_Byte *service call* to all modules. Your module can then trap this service call and implement the new OS_Byte. For full details, see the chapter entitled *Modules*.

The *FX command

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

```
*FX <reason code>[[,] <r1> [,] <r2>]]
```

The command is followed by one, two or three parameters, which may be separated by spaces or commas. The values `reason code`, `r1` and `r2` are loaded into register R0, R1 and R2 respectively; then OS_Byte is called. Any omitted values are set to zero. So:

```
MOV   R0,#218
MOV   R1,#0
MOV   R2,#255
SWI   OS_Byte
```

has the same effect as:

```
*FX 218,0,255
```

Calling *FX

The *FX command does not display any returned values; you cannot use it to read the values of status variables from the command line. It is called in the same way as any other * Command; see the chapter entitled ** Commands and the CLI* for details.

OS_Byte (SWI &06)

General purpose call to alter status variables, and perform other actions

On entry

R0 = OS_Byte number (so for OS_Byte 1, R0 = 1)
R1, R2 – as required by individual OS_Byte

On exit

R0 preserved
R1, R2 – as returned by individual OS_Byte

Interrupts

Interrupts are disabled by the OS_Byte decoding routine
Interrupt status is unaltered (ie remains disabled) for most values of R0
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant for some values of R0

Use

The action taken by this SWI depends on the reason code passed in R0. You should see the individual documentation of each OS_Byte for full details:

- If R0 is less than 128, then generally only R1 is used to pass further information. These calls set a status variable, and may also perform some other task. R2 is corrupted unless stated otherwise.
- If R0 is between 128 and 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.
- For calls with R0 between 166 and 255 on entry, the action is always the same. R0 acts as an index to a status variable, which is altered using the contents of R1 and R2:

  New Value = (Old Value AND R2) EOR R1

  To read the status variable, use R1 = &00 and R2 = &FF. To write to the status variable, use R1 = <new value> and R2 = &00.

Related SWIs

OS_Word (SWI &07)

Related vectors

ByteV

*FX

General purpose *Command to alter status variables, and perform other actions

Syntax

```
*FX <reason code> [[,] <r1> [[,] <r2>]]
```

Parameters

<reason code> from 0 to 255
<r1> from 0 to &FF (255)
<r2> from 0 to &FF (255)

Default is decimal, but any base may be used if specified eg `*FX 247 4_01`

Use

This command is used to alter status variables, which the computer uses to control its operation They can be either read or written to. Some *FX commands will also perform other actions closely related to the status variable that is being altered.

This command merely calls the SWI XOS_Byte, passing the reason code in R0, r1 in R1, and r2 in R2. The reason code determines which status variable is affected.

Individual *FX commands are not documented. You should instead refer to the documentation of individual OS_Bytes. For example, to see what *FX 218, ... will do, see the entry for OS_Byte 218.

Example

```
*FX 218,0,255
```

Related commands

None

Related SWIs

OS_Byte (SWI &06)

Related vectors

ByteV

OS_Word

Introduction

The OS_Word call is very similar to the OS_Byte call. It is also used to read from, or write to, values held in RAM by RISC OS. Much of what is said in the chapter entitled OS_Byte also applies to OS_Word.

You can add new OS_Word calls by installing a routine on the software vector that OS_Word uses – see the chapter entitled *Software vectors*. Alternatively you can use the Unknown OS_Word service call, although this is not such a good way to do so – see the chapter entitled *Modules*.

Like OS_Byte, interrupts are disabled when most OS_Word routines are entered.

The major difference between the two calls is that an OS_Word call deals with larger amounts of data than an OS_Byte call. You therefore need to pass your data in a different way.

Parameters

OS_Word always takes two parameters. R0 is a reason code (as it is for OS_Byte). R1, however, is a pointer to a parameter block. This is an area of memory where you store parameters that you want to pass to OS_Word, and where OS_Word can store its results. The size of the parameter block varies from call to call, and is documented with each OS_Word description. Often the parameter block contains a sub-reason code, which can specify the length of the parameter block; so the size can also vary for a given reason code in R0.

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_Bytes; 0 - 22 is the current range of R0 on entry. Most of these OS_Word calls are provided to ease the task of porting software from the earlier BBC and Master series computers.

Calling OS_Word

You call the OS_Word SWI in exactly the same way as any other SWI. See the earlier entitled *An introduction to SWIs* for details.

OS_Word and * Commands

Unlike OS_Byte, no * Command equivalent to OS_Word is provided.

OS_Word (SWI &07)

General purpose call to alter status variables, and perform other actions

On entry	R0 = reason code R1 = pointer to parameter block
On exit	R0 preserved
Interrupts	Interrupts are disabled by the OS_Word decoding routine Interrupt status is unaltered (ie remains disabled) for most values of R0 Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	The action taken by this SWI depends on the reason code passed in R0. In general, OS_Word is used to either read or write a large number of status variables at once. R1 points to a parameter block, the length of which varies depending on the reason code. You should see the individual documentation of each OS_Word for full details.
Related SWIs	OS_Byte (SWI &06)
Related vectors	WordV

Software vectors

Introduction

We have already seen that one of the most important features of RISC OS is the ease with which it can be altered and extended. Most of RISC OS is written as modules; these can be replaced, and extra ones can be added.

The exception to this is the kernel, which provides the central core of functions necessary for RISC OS to work. You cannot replace the entire kernel. Instead, you can change or replace how certain fundamental routines of the RISC OS kernel work. You do this by using *software vectors*, or *vectors* for short. These are held in the computer's RAM; RISC OS uses them to record where it can find these routines.

Many of these routines perform all the functions of a given SWI. The corresponding SWI is then known as a *vectored SWI*.

Claiming vectors

When you call a SWI, RISC OS uses the SWI number to decide which routine in the RISC OS ROMs you want. For an ordinary SWI, RISC OS looks up the address of the SWI routine and then branches to it. However, if you call a vectored SWI, it instead gets the address from the corresponding vector that is held in RAM. Normally this would be the address of the standard routine held in ROM.

You can change this address by using the SWI OS_Claim, documented later in this chapter. RISC OS will then instead branch to your own routine, held at the address you pass to OS_Claim.

Your own routine can do one of the following:

- replace the original routine, passing control directly back to the caller
- do some processing before calling the standard routine, which then passes control back to the caller
- call the standard routine, process some of the results it returns, and then pass control back to the caller.

If your routine completely replaces the standard one, it is said to *intercept* the call; otherwise it is said to *pass on* the call.

An example

As an example, let's look at the OS_WriteC routine. When RISC OS decodes a SWI with SWI number &00, it knows that you are requesting a write character operation. RISC OS gets an address from a vector – in this case called WrchV – and passes control to the routine.

Now by default, the WrchV contains the address of the standard write character routine in ROM. If you claim the vector using OS_Claim, whenever an OS_WriteC is executed, your own routine will be called first.

Vector chains

So far, we've deliberately been vague about how vectors store the addresses of the routine. In fact, the vector is the head of a chain of structures, which point to the next claimant on the vector, and to both the code and the data associated with this claimant. Consequently:

- there may be more than one routine on a given vector
- no claimant has to remember what the previous owner of the vector was
- vectors can be claimed and released by many different pieces of software in any order, not just in a stack-like order.

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.

When not to intercept a vector

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.

Multiply installing the same routine

When OS_Claim adds a routine to a vector, it automatically removes any earlier instances of the same routine from the chain. If you don't want this to happen, use the SWI OS_AddToVector instead.

OS_Claim (SWI &1F)

Adds a routine to the list of those that claim a vector

On entry

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

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered as it disables IRQ

Use

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

Any earlier instances of the same routine are removed. Routines are defined to be the same if the values passed in R0, R1 and R2 are identical.

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"
```

Related SWIs

OS_Release (SWI &20), OS_CallAVector (SWI &34),
OS_AddToVector (SWI &47)

Related vectors

All

OS_Release (SWI &20)

Removes a routine from the list of those that claim a vector

On entry

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

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered as it disables IRQ

Use

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"
```

Related SWIs

OS_Claim (SWI &1F), OS_CallAVector (SWI &34),
OS_AddToVector (SWI &47)

Related vectors

All

OS_CallAVector (SWI &34)

Calls a vector directly

On entry

R0 - R8 = vector routine parameters
R9 = vector number
V and C flags in R15 = flags to pass to vector

On exit

Dependent on vector called

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant – but not all vectors it calls are re-entrant

Use

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 some calls tp RemV or CnpV. It is also used by system extensions such as the Draw, ColourTrans and Econet modules to call their corresponding vectors.

You must not use this SWI to call ByteV and other such vectors, as the vector handlers expect entry conditions you may not provide.

Related SWIs

OS_Claim (SWI &1F), OS_Release (SWI &20),
OS_AddToVector (SWI &47)

Related vectors

All

OS_AddToVector (SWI &47)

Adds a routine to the list of those that claim a vector

On entry

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

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered as it disables IRQ

Use

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

Unlike OS_Claim, any earlier instances of the same routine remain on the vector chain.

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.

Related SWIs

OS_Claim (SWI &1F), OS_Release (SWI &20),
OS_CallAVector (SWI &34)

Related vectors

All

Use of registers

If you write a routine that uses a vector, it must obey the same entry and exit conditions as the corresponding RISC OS routine. For example, a routine on WrchV must preserve all registers, just as the SWI OS_WriteC does.

If you pass the call on, you can deliberately alter some of the registers to change the effect of the call. However. if you do so, you must arrange for control to return again to your routine. You must then restore the register values that the old routine would normally have returned, before finally returning control to the calling program. This is because some applications might rely on the returned values being those documented in this manual.

Processor modes

The processor mode in which the routine is entered depends on the vector:

- routines vectored through IrqV (Vector &02) are always executed in IRQ mode
- routines vectored through EventV, InsV, RemV, CnpV (Vectors &10 - &16) and TickerV (Vector &1C) are generally executed in IRQ mode, but may be executed in SVC mode if called using OS_CallAVector, and in certain other unspecified circumstances
- all other routines are executed in SVC mode – the mode entered when the SWI instruction is executed.

SVC mode

Note that if you call a SWI from a routine that is in SVC mode, you will corrupt the return address held in R14. Consequently, your routine should use the full, descending stack addressed by R13 to save R14 first. See the earlier chapter entitled *An introduction to SWIs* for a more complete explanation of this.

IRQ mode

If your routine will be entered in IRQ mode there are other restrictions. These are detailed in full in the later chapter entitled *Interrupts and handling them*.

Returning errors

Routines using most of the vectors can return errors by setting the V flag, and storing an error pointer in R0. The routine must not pass on the call, as one of the parameters (R0) has been changed; this would cause problems for the next routine on the vector. The routine must instead intercept the call, returning control back to the calling program.

You can't do this with all the vectors; some of them (those involving IRQ calls in particular) have nowhere to send the error to.

Returning from a vectored routine

You should use one of two methods to return from a vectored routine.

Passing on the call

If you wish to pass on the call (to the previous owner), you should return by copying R14 into the PC. Use the instruction:

```
MOVS PC,R14
```

When you pass on a call, you must preserve the V and C flags for the next routine. Note especially that the CMP instruction corrupts these flags; arrange your code to instead use the TEQ instruction with unshifted operands.

Intercepting the call

If you wish to intercept the call, you should pull an exit address (which has been set up by RISC OS) from the stack and jump to it. Use the instruction:

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

Control will return to the caller of the vector.

List of software vectors

The software vectors are listed below. The names of the routines which can cause the vector to be called are in brackets:

Vector	No	Description
UserV	(&00)	User vector (reserved)
ErrorV	(&01)	Error vector (OS_GenerateError)
IrqV	(&02)	Interrupt vector
WrchV	(&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 (OS_Byte)
WordV	(&07)	OS_Word indirection vector (OS_Word)
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 (OS_Byte)
RemV	(&15)	Buffer remove vector (OS_Byte)
CnpV	(&16)	Count/Purge Buffer vector (OS_Byte)
UKVDU23V	(&17)	Unknown VDU23 vector (OS_WriteC)
UKSWIV	(&18)	Unknown SWI vector (SWI)
UKPLOTV	(&19)	Unknown VDU25 vector (OS_WriteC)
MouseV	(&1A)	Mouse vector (OS_Mouse)
VDUXV	(&1B)	VDU vector (OS_WriteC)
TickerV	(&1C)	100Hz pacemaker vector
UpcallV	(&1D)	Warning vector (OS_UpCall)
ChangeEnvironmentV	(&1E)	Environment change vector (OS_ChangeEnvironment)
SpriteV	(&1F)	OS_SpriteOp indirection vector
DrawV	(&20)	Draw SWI vector (Draw_...)
EconetV	(&21)	Econet activity vector (Econet_...)
ColourV	(&22)	ColourTrans SWI vector (ColourTrans_...)

Summary of vectors

Brief details of these vectors are given below.

Many of them are by default used to indirect calls of SWIs, and so the routine they call is the same as that the SWI calls. In these cases, you should see the description of the SWI for details of entry and exit conditions. Vectors which do not have corresponding SWIs are instead documented in more detail later in this chapter.

As an example, the default routine called by WrchV is the same as that used by OS_WriteC, and so you should see the description of OS_WriteC for details of it.

About the filing system vectors

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

Other vectors and resets

Vectors are freed on any kind of reset, and system extension modules must claim them again if they need to – just as FileSwitch does.

UserV

UserV is a reserved vector. Its default action is to do nothing.

ErrorV

ErrorV is used to indirect all errors from error-generating SWIs and from OS_GenerateError – see its entry for full details. The default action is to call the error handler.

See also the rest of the chapter entitled *Generating and handling errors*; and the chapter entitled *Program Environment* for more about handlers.

IrqV

IrqV is called when an unknown IRQ is detected. It enables you to add interrupt generating devices of your own to the computer. The default action is to disable the interrupting device. See later in this chapter for full details.

See also the chapter entitled *Interrupts and handling them*, and the chapter entitled *Program Environment* for more about handlers.

WrchV

WrchV is used to indirect all calls to OS_WriteC – see its entry for full details. The default action is to call the ROM write character routine.

RdchV

RdchV is used to indirect all calls to OS_ReadC – see its entry for full details. The default action is to call the ROM read character routine.

CLIV

CLIV is used to indirect all calls to OS_CLI – see its entry for full details. The default action is to call the ROM command line interpreter.

ByteV

ByteV is used to indirect all calls to OS_Byte – see its entry for full details. The default action is to call the ROM OS_Byte routine.

Note that interrupts are disabled when an OS_Byte is called. If you claim this vector, your routine must enable interrupts if its processing takes a long time (over 100μs).

WordV

WordV is used to indirect all calls to OS_Word – see its entry for full details. The default action is to call the ROM OS_Word routine.

Note that interrupts are disabled when an OS_Word is called. If you claim this vector, your routine must enable interrupts if its processing takes a long time (over 100µs).

FileV

FileV is used to indirect all calls to OS_File – see its entry for full details. The default action is to call the ROM OS_File routine (see the note above).

ArgsV

ArgsV is used to indirect all calls to OS_Args – see its entry for full details. The default action is to call the ROM OS_Args routine (see the note above).

BGetV

BGetV is used to indirect all calls to OS_BGet – see its entry for full details. The default action is to call the ROM OS_BGet routine (see the note above).

BPutV

BPutV is used to indirect all calls to OS_BPut – see its entry for full details. The default action is to call the ROM OS_BPut routine (see the note above).

GBPBV

GBPBV is used to indirect all calls to OS_GBPB – see its entry for full details. The default action is to call the ROM OS_GBPB routine (see the note above).

FindV

FindV is used to indirect all calls to OS_Find – see its entry for full details. The default action is to call the ROM OS_Find routine (see the note above).

ReadLineV

ReadLineV is used to indirect all calls to OS_ReadLine – see its entry for full details. The default action is to call the ROM OS_ReadLine routine.

FSCV

FSCV is used to indirect calls to OS_FSControl – see its entry for full details. The default action is to call the ROM OS_FSControl routine.

EventV

EventV is used to indirect all calls to OS_GenerateEvent – see its entry for full details. The default action is to call the event handler.

See also the rest of the chapter entitled *Events*; and the chapter entitled *Program Environment* for more about handlers.

InsV

InsV is called to place a byte in a buffer. See later in this chapter for full details.

See also the chapter entitled *Buffers*.

RemV

RemV is called to remove a byte from a buffer. See later in this chapter for full details.

See also the chapter entitled *Buffers*.

CnpV

CnpV is called to count the number of entries in a buffer, or to purge the contents of a buffer. See later in this chapter for full details.

See also the chapter entitled *Buffers*.

UKVDU23V

UKVDU23V is called when a VDU 23,*n* command is issued with an unknown value of *n*. The default action is to do nothing – unknown VDU 23s are usually ignored. See later in this chapter for full details.

UKSWIV

UKSWIV is called when a SWI is issued with an unknown SWI number. The default action is to call the unknown SWI handler, which by default generates a `No such SWI` error. See later in this chapter for full details.

See also the the chapter entitled *An introduction to SWIs*; and the chapter entitled *Program Environment* for more about handlers.

UKPLOTV

UKPLOTV is called when a VDU 25,*n* (Plot) command is issued with an unknown value of *n*. The default action is to do nothing – unknown VDU 25s (Plots) are usually ignored. See later in this chapter for full details.

MouseV

MouseV is used to indirect all calls to OS_Mouse – see its entry for full details. The default action is to call the ROM OS_Mouse routine.

VDUXV

VDUXV is called when VDU output has been redirected by setting bit 5 of the OS_WriteC destination flag. This vector is normally claimed by the Font Manager, to implement the Font system. If the Font module is disabled, the default action is to do nothing – no output is sent to the VDU. See later in this chapter for full details.

See also the chapters *Character output*, *VDU drivers* and *The Font manager*.

TickerV

TickerV is called every centi-second. It must never be intercepted. See later in this chapter for full details.

UpCallV

UpCallV is used to indirect all calls to OS_UpCall – see its entry for full details. The default action is to call the UpCall handler.

ChangeEnvironmentV

ChangeEnvironmentV is used to indirect all calls to OS_ChangeEnvironment – see its entry for full details. The default action is to call the ROM OS_ChangeEnvironment routine.

SpriteV

SpriteV is used to indirect all calls to OS_SpriteOp – see its entry for full details. The default action is to call the ROM OS_SpriteOp routine.

DrawV

DrawV is used to indirect all SWI calls made to the Draw module. The default action is to call the ROM routine in the Draw module that decodes and executes SWIs. See later in this chapter for full details.

See also the chapter entitled *Draw module*.

EconetV

EconetV is called whenever there is activity on the Econet. The default action is to display the Hourglass on the screen. See later in this chapter for full details.

See also the chapters entitled *Econet*, *Hourglass* and *NetStatus*.

ColourV

ColourV is used to indirect all SWI calls made to the ColourTrans module. The default action is to call the routine in the ColourTrans module that decodes and executes SWIs. See later in this chapter for full details.

See also the chapter entitled *ColourTrans*.

Vector descriptions

The next section describes in detail those vectors which do more than indirecting a single RISC OS SWI.

In most cases, the interrupt status is given as *undefined*. This is because the vectors may be called either by the SWI(s) which normally use them, many of which ensure a given interrupt status, or by OS_CallAVector, which does not alter the interrupt status.

IrqV (Vector &02)

Called when an unknown IRQ is detected

On entry	No parameters passed in registers
On exit	—
Interrupts	Interrupts are disabled Fast interrupts are enabled
Processor mode	Processor is in IRQ mode

Use

This vector is called when an unknown IRQ is detected.

It was provided in the Arthur operating system so you could add interrupt generating devices of your own to the computer. RISC OS provides a new method of doing so that is more efficient, which you should use in preference. This vector has been kept for compatibility.

The default action is to disable the interrupt generating device by masking it out in the IOC chip.

Routines that claim this vector must not corrupt any registers. You must not call this vector using OS_CallAVector

You must intercept calls to this vector and service the interrupt if the device is yours. You must pass them on to earlier claimants if the device is not yours, so that interrupt handlers written to run under Arthur can still trap interrupts they recognise.

Old software that handled Sound interrupts using this vector will no longer work, as the new Sound module exclusively uses the RISC OS SoundIRQ device handler.

See the chapter entitled *Interrupts and handling them* for details of how to add interrupt generating devices to your computer, and the chapter entitled *Program Environment* for more about handlers.

Related SWIs

None

InsV (Vector &14)

Called to place a byte in a buffer

On entry

R0 = byte to be inserted
R1 = buffer number

On exit

R0, R1 preserved
R2 corrupted
C = 1 implies insertion failed

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Use

This vector is called by OS_Byte 138 and OS_Byte 153. The default action is to call the ROM routine to insert a byte into a buffer from the system buffers. To use different sized buffers, you must provide handlers for all of InsV, RemV and CnpV.

It may also be called using OS_CallAVector. It must be called with interrupts disabled (the OS_Bytes do this automatically), therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.

See also the entitled *Buffers*.

Related SWIs

OS_Byte 138 and 153 (SWI &06)

RemV (Vector &15)

Called to remove a byte from a buffer

On entry

R1 = buffer number
V flag = 1 if buffer to be examined only, else V flag = 0

On exit

R0 = next byte to be removed (for examine option)
R1 preserved
R2 = byte removed (for remove option)
C = 1 means buffer was empty on entry

'nterrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Use

This vector is called by OS_Byte 145 and OS_Byte 152. The default action is to call the ROM routine to inspect or remove a byte from the system buffers. To use different sized buffers, you must provide handlers for all of InsV, RemV and CnpV.

It may also be called using OS_CallAVector. It must be called with interrupts disabled (the OS_Bytes do this automatically), therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.

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

See also the entitled *Buffers*.

Related SWIs

OS_Byte 145 and 152 (SWI &06)

CnpV (Vector &16)

Called to count the number of entries in a buffer, or to purge the contents of a buffer

On entry

R1 = buffer number
V flag = reason code
C = return value flag, if V flag = 0

On exit

R0 corrupted
R1 = least significant 8 bits of count, if V flag = 0 on entry
R2 = most significant 24 bits of count, if V flag = 0 on entry
R1, R2 preserved if V flag = 1 on entry

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Use

This vector is called by OS_Byte 15, OS_Byte 21 and OS_Byte 128. The default action is to call the ROM routine to count the number of entries in a buffer, or to purge the contents of a buffer.

It may also be called using OS_CallAVector. It must be called with interrupts disabled (the OS_Bytes do this automatically), therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.

The V flag gives a reason code that determines the operation:

V flag = 0	count the entries in a buffer
V flag = 1	purge the buffer

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

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

See also the entitled *Buffers*.

Related SWIs

OS_Byte 15, 21 and 128 (SWI &06)

UKVDU23V (Vector &17)

Called when an unrecognised VDU 23 command is issued.

On entry

R0 = VDU 23 option requested
R1 = pointer to VDU queue

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

This vector is called when a VDU 23,*n* command is issued with an unknown value of *n*, 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 – unknown VDU 23s are ignored.

Related SWIs

None

UKSWIV (Vector &18)

Called when an unknown SWI instruction is issued

On entry

R0 - R8 as set up by the caller
R11 = SWI number

On exit

Generates an error by default

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

This vector is called when a SWI is issued with an unknown SWI number. 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, which by default returns a `No such SWI` error. See later in this for full details.

This vector can be used to add large numbers of SWIs to the system from a single module. Normally only 64 SWIs can be added by a module; if you claim this vector, you can then trap any additional SWIs you wish to add. (You should always use the module mechanism to add the first 64 SWIs that a module adds, as it is more efficient than using this vector.) Note that you must get an allocation of SWI numbers from Acorn before adding any to commercially available software. This will avoid clashes between your own software and other software.

See also the the entitled *An introduction to SWIs*; and the chapter entitled *Program Environment* for more about handlers.

Related SWIs

OS_UnusedSWI (SWI &19)

UKPLOTV (Vector &19)

Called when an unknown PLOT command is issued

On entry

R0 = PLOT number

On exit

R0 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

This vector is called by the VDU drivers when a VDU 25,*n* (PLOT) or SWI OS_Plot command is issued with an unknown value of *n*.

By using OS_ReadVduVariables you can read the co-ordinates of the last three points that have been visited, and the one specified in the unknown PLOT command. These are held in the VDU variables 138 - 147. See the entry for OS_ReadVduVariables for full details.

When the call returns to the VDU drivers they 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 – unknown VDU 25s (Plots) are ignored.

Related SWIs

None

VDUXV (Vector &1B)

Called when VDU output has been redirected

On entry

R0 = byte sent to the VDU

On exit

R0 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

This vector is called when VDU output has been redirected by setting bit 5 of the OS_WriteC destination flag. 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.

This vector is normally claimed by the Font Manager, to implement the Font system. If the Font module is disabled, the default action is to do nothing – no output is sent to the VDU.

See also the chapters *Character output*, *VDU drivers* and *The Font manager*.

Related SWIs

None

TickerV (Vector &1C)

Called every centi-second

On entry	No parameters passed in registers
On exit	—
Interrupts	Interrupts are disabled Fast interrupts are enabled
Processor mode	Processor is in IRQ or SVC mode
Use	This vector is called every centi-second. It must never be intercepted, as this might prevent other users from being called. Routines that take a long time (say > 100µs) may re-enable IRQ so long as they disable it again before passing the call on. If you do so, other calls may be made to TickerV in the meantime. Your routine needs to prevent re-entrancy. One way of doing so is: • to use a flag in its workspace to note that it is currently threaded, and: • to keep a count of how many calls to TickerV have been missed while it was threaded, so the count can be examined on exit and corrected for.
Related SWIs	None

DrawV (Vector &20)

Used to indirect all SWI calls made to the Draw module

On entry	R0 - R7 dependant on SWI issued R8 = index of SWI within the Draw module SWI chunk
On exit	Dependant on SWI issued
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Use	This vector is used to indirect all SWI calls made to the Draw module. The default action is to call the ROM routine in the Draw module that decodes and executes SWIs.

The index held in R8 is decoded as follows:

0	Draw_ProcessPath
2	Draw_Fill
4	Draw_Stroke
6	Draw_StrokePath
8	Draw_FlattenPath
10	Draw_TransformPath

See also the chapter entitled *Draw module*.

Related SWIs

Draw_... (SWIs &40700 - &4073F)

EconetV (Vector &21)

Called whenever there is activity on the Econet

On entry

R0 = reason code
R1 = total size of data, or amount of data transferred, or no parameter passed

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

EconetV is called whenever there is activity on the Econet. The reason code tells you what the activity is.

The bottom nibble of the reason code indicates whether the activity has started (0), is part way through (1) or finished (2). The next nibble gives the type of operation.

The table below shows the reason codes that are passed. The right hand column shows what is passed in R1, or (for the less obvious cases) when the reason code is passed:

&10	NetFS_StartLoad	R1 = total size of data
&11	NetFS_PartLoad	R1 = amount of data transferred
&12	NetFS_FinishLoad	
&20	NetFS_StartSave	R1 = total size of data
&21	NetFS_PartSave	R1 = amount of data transferred
&22	NetFS_FinishSave	
&30	NetFS_StartCreate	R1 = total size of data
&31	NetFS_PartCreate	R1 = amount of data transferred
&32	NetFS_FinishCreate	
&40	NetFS_StartGetBytes	R1 = total size of data
&41	NetFS_PartGetBytes	R1 = amount of data transferred
&42	NetFS_FinishGetBytes	
&50	NetFS_StartPutBytes	R1 = total size of data
&51	NetFS_PartPutBytes	R1 = amount of data transferred

&52	NetFS_FinishPutBytes	
&60	NetFS_StartWait	start of a Broadcast_Wait
&62	NetFS_FinishWait	end of a Broadcast_Wait
&C0	Econet_StartTransmission	start to wait for a transmission to end
&C2	Econet_FinishTransmission	DoTransmit returns
&D0	Econet_StartReception	start to wait for a reception to end
&D2	Econet_FinishReception	WaitForReception returns

This vector is normally claimed by the NetStatus module, which uses the Hourglass module to display an hourglass while the Econet is busy. It passes on the call. If the Hourglass module is disabled, the default action is to do nothing. See the chapters entitled *Hourglass* and *NetStatus*.

See also the chapters entitled *NetFS*, *NetPrint* and *Econet*.

Related SWIs

NetFS_... (SWIs &40040 - &4007F),
Econet_... (SWIs &40000 - &4003F),
NetPrint_... (SWIs &40200 - &4023F) and
Hourglass_... (SWIs &406C0 - &406FF)

ColourV (Vector &22)

Used to indirect all SWI calls made to the ColourTrans module

On entry	R0 - R7 dependant on SWI issued R8 = index of SWI within the ColourTrans module SWI chunk
On exit	Dependant on SWI issued
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Use	This vector is used to indirect all SWI calls made to the ColourTrans module. The default action is to call the routine in the ColourTrans module that decodes and executes SWIs. See also the chapter entitled *ColourTrans*.
Related SWIs	ColourTrans_... (SWIs &40740 - &4077F)

More complex uses of vectors

Sometimes, you may want to do more complex things with a vector, such as:

- preprocessing registers to alter the effect of a standard routine
- postprocessing to change the effect of future calls
- repeatedly calling a routine or group of routines.

There are a number of important things to remember if you are doing so. You must make sure that:

- the vector still looks exactly the same to a program that is calling it, even if it now does different things
- your routine will cope with being called in all the processor modes that its vector uses (for example, SVC or IRQ mode for a routine on InsV)
- the values of R10 and R11 are preserved when earlier claimants of the vector are repeatedly called.

An example program

The example program below illustrates all these important points. You can adapt it to write your own routines.

The program claims WrchV, adding a routine that:

- changes the case of the character depending on the state of a flag (pre-processing)
- calls the remaining routines on the vector to write the altered character
- toggles the flag (post-processing)
- ensures that all registers are set to the values that would be returned by the default write character routine
- returns control to the calling program.

Note that the program releases the vector before ending, even if an error occurs.

```
DIM code% 100
FOR pass%=0 TO 3 STEP 3
P%=code%
[ OPT pass%
.vectorcode%
; save the entry value, the necessary state for the repeated call,
; and our workspace pointer
    STMFD   r13!, {r0, r10-r12, r14}
```

```
; do our preprocessing; as a trivial example, convert to the current case
    LDRB    r14, [r12]          ; pick up upper/lowercase flag
    CMP     r14, #0
    BEQ     uppercase%
    CMP     r0, #ASC"A"         ; lowercase the character
    RSBGES  r14, r0, #ASC"Z"
    ADDGE   r0, r0, #ASC"a"-ASC"A"
    B       done_preprocess%

.uppercase%
    CMP     r0, #ASC"a"         ; uppercase the character
    RSBGES  r14, r0, #ASC"z"
    SUBGE   r0, r0, #ASC"a"-ASC"A"

.done_preprocess%

; now do the call to the rest of the vector. Since this is WrchV, we know that
; we are in SVC mode; however, the code below will correctly call the rest of
; the vector whatever the mode.

    STMFD   r13!, {r15}         ; pushes PC+12, complete with flags and mode
    ADD     r12, r13, #8        ; stack contains pc,r0,r10,r11,r12,r14
                                ; so point at the stacked r10
    LDMIA   r12, {r10-r12, r15} ; and restore the state needed to call the
                                ; rest of the chain (r10 and r11), and
                                ; "return" to the non-vector claiming address.
                                ; The load of r12 wastes one cycle.

  ; we are now at the pc+12 that we stacked; this is therefore where the
  ; rest of the vector returns to when it has finished.

    LDR     r12, [r13, #12]     ; reload our workspace pointer
                                ; Note that the offset of #12 (and the earlier
                                ; #8 when we pushed onto the stack) refer to
                                ; this example only and are not general
                                ; Note also that the pc we pushed was
                                ; pulled by the vector claimer.

  ; we could now do some more processing, set r0 up to another character,
  ; and loop round to done_preprocess% again; instead, we'll just do some
  ; example postprocessing; we'll toggle our upper/lowercase flag.

    LDRB    r14, [r12]
    EOR     r14, r14, #1
    STRB    r14, [r12]

  ; now return; if there was no error then intercept the call to the
  ; vector, returning the original character.

    LDMVCFD r13!, {r0, r10-r12, r14, r15}

  ; could pass the call on instead by omitting r14 from the addresses
  ; to pull - ie use LDMVCFD r13!, {r0, r10-r12, r15}

  ; there was an error; set up the correct error pointer, flags, and
  ; claim the vector.

    STR     r0, [r13]           ; save the error pointer
    LDMFD   r13!, {r0, r10-r12, r14, r15}
                                ; return with V still set, and claim the vector
]
NEXT
DIM flag% 1
?flag%=0
WrchV%=3
ON ERROR SYS "XOS_Release", WrchV%, vectorcode%, flag%: PRINTREPORT$:END
SYS "OS_Claim", WrchV%, vectorcode%, flag%
REPEAT
  INPUT command$
  OSCLIcommand$
UNTILcommand$=""
SYS "XOS_Release", WrchV%, vectorcode%, flag%
END
```

Hardware vectors

Introduction

The hardware vectors are a set of words starting at logical address &0000000. The ARM processor branches to these locations in certain exceptional conditions – in general, either when a privileged mode is entered or when a hardware error occurs. These conditions are known as *exceptions*. Usually, each vector will contain a branch to a routine to handle the exception. The vectors, their addresses and their default contents are:

Addr	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...`	

Reset vector

When the computer is reset, amongst other things:

- the ROM is temporarily switched into location zero
- the program counter is loaded with &00.

The reset vector is hence read from the ROM and will always be the same.

Any attempt to jump to location zero in RAM will result in a `Branch through zero` error.

Hardware exception vectors

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 *Program Environment*. Very few programs need to take account of these vectors.

The usual action of these exceptions is to cause an error. The default handlers for these exceptions also dump the aborting mode's registers into the current ExceptionDumpArea, and test to see if the exception occurred while the processor was in FIQ mode. If it was then FIQs are disabled on the IOC chip so that the exception does not recur – this would overwrite the original register dump, and probably hang the machine.

Undefined instruction vector

The undefined instruction vector is called when the ARM attempts to execute an instruction that is not a part of its normal instruction set. Before calling this vector, the ARM is forced to SVC mode, and interrupts are disabled. If the floating point emulator (either hardware or software) is active, it intercepts the undefined instruction vector to interpret floating point instructions, and passes on those that it does not recognise.

Prefetch abort vector

The prefetch abort vector is called when the MEMC chip detects an illegal attempt to prefetch an instruction. There are two possible reasons for this:

- an attempt was made to access protected memory from an insufficiently privileged mode
- an attempt was made to access a non-existent logical page.

Before calling this vector, the ARM is forced to SVC mode, and interrupts are disabled.

Data abort vector

The data abort vector is called when the MEMC chip detects an illegal attempt to fetch data. There are two possible reasons for this:

- an attempt was made to access protected memory from an insufficiently privileged mode
- an attempt was made to access a non-existent logical page.

Before calling this vector, the ARM is forced to SVC mode, and interrupts are disabled.

Address exception vector

The address exception vector is called when a data reference is made outside the range 0 - &3FFFFFF. Before calling this vector, the ARM is forced to SVC mode, and interrupts are disabled.

SWI vector

The SWI vector is called when a SWI instruction is issued. It contains a branch to the RISC OS code which decodes the SWI number and branches to the appropriate location. Before calling this vector, the ARM is forced to SVC mode, and interrupts are disabled.

You are strongly recommended not to replace this vector.

For full details, see the earlier chapter entitled *An introduction to SWIs*.

IRQ vector

The IRQ vector is called when the ARM receives an interrupt request. It also contains a branch into the RISC OS code. This code attempts to deal with the interrupt by examining the IOC chip, to find the highest priority device that has interrupted the processor. If no interrupting device is found then the software vector IrqV is called.

Before calling the hardware IRQ vector, the ARM is forced to IRQ mode, and interrupts are disabled.

For full details, see the chapter entitled *Interrupts and handling them.*

FIQ vector

Finally, the FIQ vector is called when the ARM receives a fast interrupt request. For some claimants (such as ADFS) this is the first instruction of a RAM-based routine to deal with the fast interrupt requests. For other claimants (such as NetFS) this is a branch instruction to the code that deals with the fast interrupt requests. (NetFS uses FIQs to drive a state machine, so the overhead of copying code to the FIQ vector is much more than that of putting a Branch instruction there.)

Before calling this vector, the ARM is forced to FIQ mode, and both normal and fast interrupts are disabled.

For full details, see the chapter entitled *Interrupts and handling them.*

Claiming hardware vectors

If you are the current application, you can change the effect of most of the hardware vectors by installing the appropriate handler. If you are not, then you will have to 'claim' the vector yourself. This is most likely to occur if you are a system extension module. There is no SWI to claim a hardware vector; instead you have to overwrite it with a `B myHandler` or a `LDR PC, [PC, #myHandlerOffset]` instruction, and do all the 'housekeeping' yourself.

Passing on calls to hardware vectors

You must make sure that if your own handler cannot process what caused the vector to be called, the 'next' handler for the vector is called. The address of the next handler can be dynamic, so you must be careful:

- If the instruction in the hardware vector location when you come to claim it is `B oldHandler`, then you need to compute the address of the old handler and store it in your workspace. You then need to store a pointer to this address.
- If the instruction is `LDR [PC, #oldHandlerOffset]`, then you need to compute the address of the variable where RISC OS stores the installed handler's address, and store this pointer. You **must not** dereference this pointer to get the actual address of the handler, as this value may change as different applications are run.

In both cases above you now have a pointer to a variable which holds the address of the next handler to call; you can then use identical code in both cases to pass on a call to the hardware vector that you cannot handle.

Releasing hardware vectors

If your module is killed, so you need to release a hardware vector, you must first check to see that the instruction that is in the hardware vector location points to your own handler. If it does not, your module must refuse to die, as another piece of software has stored away the address of your handler, and may try to pass on a call to your handler or to restore you when it exits.

Vector priorities

The hardware vectors have different priorities, so that if exceptions occur simultaneously they are sensibly handled. This list shows the vectors in order of priority:

Reset	
Address exception	
Data abort	↑ High priority
FIQ	
IRQ	
Prefetch abort	↓ Low priority
Undefined instruction	
SWI	

Interrupts and handling them

Introduction

An *interrupt* is a signal sent to the ARM processor from a hardware device, indicating that the device requires attention. They are sent, for example, when a key has been pressed or when one of the software timers needs updating. This sending of a signal is known as an *interrupt request*.

RISC OS deals with the interrupt by temporarily halting its current task, and entering 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.

Interrupts provide a very efficient means of control since the processor doesn't have to be responsible for regularly checking to see if any hardware devices need attention. Instead, it can concentrate on executing your code or whatever else its current main task may be, and only deal with hardware devices when necessary.

Devices handled

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

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

Expansion cards

Additionally, external hardware such as expansion cards may cause new interrupts to be generated. For example, the analogue to digital convertor on the BBC I/O expansion card can interrupt when it has finished a conversion. It is therefore possible to install routines which deal with these new interrupts.

Device numbers

Each potential source of interrupts has a *device number*. There are corresponding *device vectors*; installed on each vector there is a default *device driver* that receives only the interrupts from that device.

Unless you are adding your own interrupt-generating devices to the computer, you should not need to alter the interrupt system.

The device numbers are:

0	Printer Busy
1	Serial port Ringing Indicator
2	Printer Acknowledge
3	VSync Pulse
4	Power On – this should never appear
5	IOC Timer 0
6	IOC Timer 1
7	FIQ Downgrade – reserved for the use of the current owner of FIQ
8	Expansion card FIQ Downgrade – this should normally be masked off
9	Sound system buffer change
10	Serial port controller
11	Hard disc controller
12	Floppy disc changed
13	Expansion card
14	Keyboard serial transmit register empty
15	Keyboard serial receive register full

The device numbers correspond to bits of the interrupt registers held in the IOC chip – see the end of this chapter for more details.

Device vectors

Just like other vectors in RISC OS, you can claim the device vectors and get them to call a different routine. You do this using the SWI OS_ClaimDeviceVector.

Most of the device vectors only call the most recent routine that claimed the vector. There is no mechanism to pass on the call to earlier claimants, as it is not sensible to have many routines handling one device. However, old claimants remain on the vector, and if you release the vector using OS_ReleaseDeviceVector, the previous owner of the vector is re-installed.

The exceptions to this are device vectors 8 and 13, which handle FIQs and IRQs (respectively) which are generated by expansion cards. These can have many routines installed on them, as it is possible to add many expansion cards to the computer. Each claimant specifies exactly which interrupts it is interested in; RISC OS then ensures that only the correct routine is called.

Avoiding duplication of drivers

Note that when you claim a device vector, RISC OS will automatically remove from the chain any earlier instances of the exact same routine.

Automatic interrupt disabling

If you release a device vector, and there are no earlier claimants of that vector, RISC OS will automatically disable interrupts from the corresponding device. You must not attempt to disable the interrupts yourself. There is thus guaranteed to be a device driver for each device number that can generate interrupts.

OS_ClaimDeviceVector (SWI &4B)

Claims a device vector

On entry

R0 = device number
R1 = address of device driver routine
R2 = value to be passed in R12 when device driver is called
R3 = address of interrupt status, if R0 = 8 or 13 on entry
R4 = interrupt mask to use, if R0 = 8 or 13 on entry

On exit

R0 - R4 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call installs the device driver, the address of which is given in R1, on the device vector given in R0. If the same driver has already been installed on the vector (ie the same parameters were used to install a driver) then the old copy is removed from the vector. Note that this call does not enable interrupts from the device (cf OS_ReleaseDeviceVector).

The previous driver is added to the list of earlier claimants.

If R0 = 8 or 13 then the device driver routine is for an expansion card. R3 gives the address where the expansion card's interrupt status is mapped into memory – see the chapter entitled *Expansion Cards*. Your device driver is called if the IOC chip receives an interrupt from an expansion card, and (LDRB [R3] AND R4) is non-zero.

For all other values of R0, your driver is called if the IOC chip receives an interrupt from the appropriate device, the corresponding IOC interrupt mask bit is set, and your driver was the last to claim the vector.

Related SWIs

OS_ReleaseDeviceVector (SWI &4C)

Related vectors

None

OS_ReleaseDeviceVector (SWI &4C)

Releases a device vector

On entry

R0 = Device number
R1 = call address
R2 = R12 value
R3 = interrupt location if R0 = 8 or 13 on entry
R4 = interrupt mask if R0 = 8 or 13 on entry

On exit

R0 - R4 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call removes a driver from from the list of claimants of a device vector. The device driver is identified by the contents of the registers on entry; R0 - R3 (R0 - R5 if R0 = 8 or 13) must be the same as when the device driver was installed on the vector.

The previous owner of the vector is re-installed at the head of the chain. If there is no previous owner, then IRQs from the corresponding device are disabled.

You must not attempt to disable a device's IRQs yourself when you release its vector.

Related SWIs

OS_ClaimDeviceVector (SWI &4B)

Related vectors

None

Technical details

This section gives you more technical details of how the RISC OS interrupt system works. You should refer to it if:

- you are adding an interrupt generating device to your computer – such as an expansion card
- you wish to use Timer 1 from the IOC chip, which is not used by RISC OS
- you wish to change one of the default RISC OS device driving routines.

How a device driver is called

Interrupts are generated and the device driving routine called as follows:

1 The device that needs attention alters the status of its interrupt request pin, which is connected to the IOC chip

2 The corresponding bit of one of the IOC's interrupt status registers is set

3 The IOC's interrupt status registers are ANDed with its interrupt mask registers, and the results put in its interrupt request registers

4 If the result was non-zero (ie the device's bit was set in the mask) then an interrupt is sent to the ARM processor

5 If interrupts are enabled, the ARM saves R15 in R14_irq

6 It then forces IRQ mode by setting the M1 bit and clearing the M0 bit of R15, and disables interrupts by setting the I bit

7 The ARM then forces the PC bits of R15 to &18

8 The instruction at &18 is fetched and executed. It is a branch to the code that RISC OS uses to decode IRQs

9 RISC OS examines the interrupt request registers of the IOC chip to see which device number generated the interrupt

10 If the device number was not 8 or 13 (ie the device was not an expansion card) then RISC OS calls the last routine that claimed the corresponding device vector

If the device was an expansion card, RISC OS checks each routine on the expansion card device vector, starting with the most recent claimant. The contents of the interrupt status byte are ANDed with the mask (as passed in R3 and R4 when the routine was installed). If the result is non-zero, the routine is called; otherwise the next most recent claimant is checked.

Whatever the device number, if no routine is found to handle the interrupt then IrqV (the unknown IRQ vector) is called. By default this disables the interrupting device by clearing the corresponding bit of its interrupt mask – but the call may be claimed by routines written to work under the old Arthur operating system.

11 The device driving routine is executed and returns control.

The addresses of the IOC registers are given at the end of the chapter.

How a device driver is entered

When a routine that has claimed a device vector is called:

- the ARM is in IRQ mode with interrupts disabled
- R3 points to the base of the IOC chip memory space
- R12 has the same value as R2 had when the vector was claimed – this is usually used to point to the routine's workspace.

Your routine must:

- service the interrupt
- stop the device from generating interrupts, where necessary
- return to the kernel using the instruction `MOV PC, R14`.

In doing so, you may corrupt registers R0 - R3 and R12.

Restrictions

There are more restrictions on writing code to run under IRQ mode than there are under SVC mode. These apply to:

- speed of execution
- re-enabling interrupts
- calling SWIs
- not using certain SWIs.

Speed of execution

Interrupt handling routines must be quick to execute. This is because they are entered with interrupts disabled, so while they are running other hardware may be kept waiting. This slows the machine down considerably.

In practice, 100µs is the longest you should leave interrupts disabled. If your routine will take longer than this, try to make it shorter. If all else fails, your routine must re-enable interrupts. It should do so by clearing the I bit of R15, using for example:

```
MOV     Rtemp, PC         ; I_Bit set in PSR
TEQP    Rtemp,#I_bit      ; Note TEQ is like EOR: so clears I_Bit in PSR
```

where `I_bit` is a constant having only the I bit set. You must not use the TEQ instruction directly on the PC, as this might lock out other interrupting devices that need immediate attention before their hardware buffers overflow; for example, MIDI or the serial port.

If your routine does re-enable interrupts, it must be able to cope if a second interrupt occurs, and hence the routine being entered for a second time.

Calling SWIs

Calling SWIs from device driver routines is quite similar to calling them from SWI routines. Again the problem is that R14_svc (the return address for SWIs) may get corrupted. For example:

1 A SWI is called by a program that is running in User mode. R15 (the return address to the program) is copied to R14_svc, and the processor is put into SVC mode. The SWI routine is then entered.

2 While this routine is running, an interrupt occurs. The device driver routine calls a second SWI. The ARM enters SVC mode, and R15 is copied to R14_svc, overwriting the return address to the program. The second SWI executes.

3 Control is returned to the interrupt handler.

4 When it finishes, control passes back to the first SWI routine by loading R14_irq back into R15.

5 The first SWI routine finishes executing, and tries to return control to the program by loading R14_svc back into R15.

6 Because R14_svc was overwritten by the second SWI, control is not returned to the program; instead it passes back to the second SWI again, crashing the computer.

Recommended procedure

The solution used with device driver routines is the same as that for SWI routines. R14_svc is pushed on the stack before the SWI is called, and pulled afterwards. However, this is more complex as you have to first change from IRQ to SVC mode. The recommended way of doing so 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 to prevent contention
```

`SVC_Mode` is 3. Of course, you must preserve R8 and R9 as well, using the full descending IRQ stack.

Error handling

Interrupt handling routines must only call error-returning SWIs ('X' SWIs). If you do get an error returned to the routine, you cannot return that error elsewhere. Instead you must take appropriate action within the routine. You may also like to store an error indication, so that the next call to a SWI in the module that provides the routine (or the current call, if already threaded) will generate an error.

Re-entrancy

There are some SWIs you shouldn't call at all from an interrupt handling routine, even with the above precautions. This is because they are not *re-entrant*; that is, they can't be entered while an earlier call to them may still be in progress. One common reason for this is if the routine uses some private workspace. For example:

1. The SWI is called from a program. It stores some values in the workspace.
2. An interrupt occurs. The interrupt handling routine calls the same SWI a second time.
3. The old values in the workspace are overwritten
4. When control returns to the first instance of the SWI, the workspace is corrupted and so the routine does not work correctly.

Documentation

The documentation of each SWI clearly states if it is re-entrant – ie if you can call it from an interrupt handling routine. There are three common entries:

• re-entrant	can be used
• not re-entrant	must not be used
• undefined	the SWI's re-entrancy depends on how you call it, or it is subject to future change

In general, OS_Byte and OS_Word calls can be used. OS_WriteC and routines which use it should never be called.

Clearing interrupt conditons

Before your routine returns, it must service the interrupt – that is, give the device the attention it needs, which originally caused it to generate the interrupt. You must then clear the interrupt condition, to stop the device carrying on generating the same interrupt. How you do this depends on the device, but will usually involve accessing the hardware that is generating the interrupt. See the relevant hardware data sheets for information.

Fast interrupt requests

There are actually two classes of interrupt requests. So far we have been looking at the normal IRQ. The second type is a *fast interrupt request*, or *FIQ*. Fast interrupts are generated by devices which demand that their request is dealt with as quickly as possible. They are dealt with at a higher priority than normal IRQs.

Fast interrupts are a separate system. There are separate registers in the IOC chip, separate inputs to the chip, and a separate connection to the ARM. The ARM has a processor mode reserved for FIQs, and a hardware vector.

FIQ devices

The devices handled under FIQs are as follows:

0	Floppy disc data
1	Floppy disc interrupt
2	Econet
3	C3 pin on IOC
4	C4 pin on IOC
5	C5 pin on IOC
6	Expansion card
7	Force FIQ – this bit is always set, but usually masked out

Again, the device numbers correspond to the bits in IOC registers: these are the FIQ interrupt registers.

Similarities between FIQs and IRQs

In many ways FIQs are similar to IRQs. So FIQ routines must:

- keep FIQ and IRQ disabled while they execute – if you're taking so long that you need to re-enable them, you should be using IRQs, not FIQs

Differences between FIQs and IRQs

There are three important differences:

- FIQs must be handled more quickly
- FIQs are vectored differently
- FIQs must never call SWIs.

The default owner

When a FIQ is generated execution passes directly to code at the FIQ hardware vector. By default, the code that is installed here handles FIQs generated by the Econet module, if it is present. The Econet module is the *default owner* of the FIQ vector.

When other parts of RISC OS want to use FIQs, for example to perform a disc transfer under interrupts, they claim the vector, replace the default code, and then release the vector. RISC OS automatically re-installs the default code.

Obviously only one current FIQ owner is supported.

It is vital that you only claim the FIQ vector for the absolute minimum time necessary. For example, ADFS uses FIQs to perform disc transfers; but it releases the FIQ vector between each sector.

Using FIQs

You must follow a similar procedure if you want to use FIQs. This is the sequence you must follow:

1. Claim FIQs using the module service call OS_ServiceCall. You can claim FIQs either from the foreground, or from the background.

   To claim from the foreground, the reason code in R1 must be &0C (Claim FIQ). This service call will always succeed, but will wait for any current background FIQ process to complete.

   To claim from the background, the reason code in R1 must be &47 (Claim FIQ in background). This service call may fail, but this failure does not imply an error – merely that FIQs could not be claimed. You **must** leave your routine to allow the foreground routine to finish using FIQs and release them. You should schedule a later retry; for example with a disc, you would retry next revolution of the disc. If R1 = 0 on return, you successfully claimed the FIQ vector.

2. Set the IOC fast interrupt mask register to &00, to prevent fast interrupts while you are changing the FIQ code.

3. Poke your FIQ handling routine into addresses &1C upwards. You may use memory up to location &100 (ie the last possible instruction is at &FC).

4. Enable FIQ generation from your device.

5 Set the bit corresponding to your device in the IOC fast interrupt mask register.

6 Start your FIQ operation. You must either poll for its completion, or rely on the completion starting the finalise process in the steps below.

7 End your FIQ operation

8 Set the IOC fast interrupt mask register to zero.

9 Disable FIQ generation from your device.

10 Release FIQs using the module service call OS_ServiceCall. The reason code in R1 must be &0B (Release FIQ). It doesn't matter which way you originally claimed the FIQ hardware vector.

For full details of OS_ServiceCall, see the chapter entitled *Modules*.

How the FIQ vector is called

You may need to know in more detail how fast interrupts are generated and the FIQ hardware vector is called:

1 The device that needs attention alters the status of its fast interrupt request pin, which is connected to the IOC chip

2 The corresponding bit of one of the IOC's fast interrupt status registers is set

3 The IOCs fast interrupt status registers are ANDed with its fast interrupt mask registers, and the results put in its request registers

4 If the result was non-zero (ie the device's bit in the mask was also set) then a fast interrupt is sent to the ARM processor

5 The ARM saves R15 in R14_fiq

6 It then forces FIQ mode by clearing the M1 bit and setting the M0 bit of R15, and disables all interrupts by setting both the I bit and the F bit

7 The ARM then forces the PC bits of R15 to &1C

8 The FIQ handling routine at &1C is entered

The addresses of the IOC registers are given at the end of the chapter.

Disabling interrupts

There will be times when you want to disable interrupts. You must only do so with great 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.

SWIs provided

The easiest way to disable and re-enable interrupts from user mode is to use the SWIs provided. These are OS_IntOff and OS_IntOn. They have no entry or exit conditions, and are described in full below.

More advanced cases

To disable specific devices, or fast interrupts, you need to be in a privileged mode. The example below shows you how to use the SWI OS_EnterOS to enter SVC mode. This is described in more detail below.

Normally you won't need to do this, because RISC OS places you in a privileged mode during module initialisation, service and finalisation entries – the times you are most likely to want to disable devices, or fast interrupts.

Once you are in a privileged mode, you can disable interrupts by setting the I bit in R15. You can also disable fast interrupts by setting the F bit.

To disable specific devices you must first have disabled all interrupts. You then clear the relevant bits in any of the the IOC's interrupt mask registers. This must be done in very few (no more than five) instructions Finally, you must re-enable interrupts:

```
MOV      R2,#IOC             ; Point R2 at IOC before disabling interrupts
SWI      "OS_EnterOS"        ; Enter SVC mode
MOV      R0,PC               ; Get status in R0
ORR      R1,R0,#&0C000000    ; Set the interrupt masks
TEQP     R1,#0               ; Update PSR

...                          ; Write to IOC here in < 5 instructions, eg:

LDRB     R1,[R2,#IOCIRQMskA]
ORR      R1,R1,#Timer1Bit    ; Enable Timer1
                             ; If BIC is used instead of ORR, Timer1 is disabled
STRB     R1,[R2,#IOCIRQMskA]

...                          ; End of write to IOC

TEQP     R0,#3               ; Restore entry state and return to user mode
MOVNV    R0,R0               ; NOP to avoid contention
```

FIQs must be disabled because the mask has FIQ downgrade bits. If the current FIQ owning process alters these bits between your reading the mask and writing it, the process will not then get the IRQ that it just requested the FIQ be downgraded to.

OS_IntOn (SWI &13)

Enables interrupts

On entry	No parameters passed in registers
On exit	Registers preserved
Interrupts	Interrupt status is undefined on entry Interrupts are enabled on exit Fast interrupt status is unaltered
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is re-entrant
Use	This call enables interrupts and returns to the caller with the processor mode unchanged.
Related SWIs	OS_IntOff (SWI &14)
Related vectors	None

OS_IntOff (SWI &14)

Disables interrupts

On entry	No parameters passed in registers
On exit	Registers preserved
Interrupts	Interrupt status is undefined on entry Interrupts are disabled on exit Fast interrupt status is unaltered
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is re-entrant
Use	This call disables interrupts and returns to the caller with the processor mode unchanged.
Related SWIs	OS_IntOn (SWI &13)
Related vectors	None

OS_EnterOS (SWI &16)

Sets the processor to SVC mode

On entry	No parameters passed in registers
On exit	Registers preserved
Interrupts	Interrupt status is unaltered Fast interrupt status is unaltered
Processor mode	Processor is in SVC mode during the routine, and on exit.
Re-entrancy	SWI is re-entrant
Use	This call returns to the caller in SVC mode. The interrupt states remain unchanged.
Related SWIs	None
Related vectors	None

Hardware addresses

It will help you to use interrupts to their full potential if you have a good knowledge of the hardware used to build the computer. We don't have the space to give you full details of every RISC OS computer built by Acorn in this manual.

Below we tell you where the IOC chip and some of the various peripheral controllers of a RISC OS computer are mapped into memory on an Archimedes computer. Although these may be taken as typical of RISC OS computers, there is no guarantee that other computers will be similarly mapped. Indeed, even the details below are subject to change; the peripheral controllers may be changed as improved ones become available, or the mapping may be redefined.

Always use defined software interfaces in preference to directly accessing the hardware.

Finding out more

If you need to know more, you can:

- refer to the earlier chapter entitled *ARM Hardware*
- consult the *VL86C010 32-Bit RISC MPU and Peripheral User's Manual*, published by Prentice Hall
- consult the datasheets for the various peripheral controllers used, available from their manufacturers
- contact Acorn Customer Support and Services.

IOC registers

The IOC registers are a single byte wide, and are mapped into memory like this:

Address	Read	Write
&3200000	Control	Control
&3200004	Kbd serial receive	Kbd serial transmit
&3200008	—	—
&320000C	—	—
&3200010	IRQ status A	—
&3200014	IRQ request A	IRQ clear
&3200018	IRQ mask A	IRQ mask A
&320001C	—	—
&3200020	IRQ status B	—
&3200024	IRQ request B	—
&3200028	IRQ mask B	IRQ mask B
&320002C	—	—
&3200030	FIQ status	—
&3200034	FIQ request	—
&3200038	FIQ mask	FIQ mask
&320003C	—	—
&3200040	T0 count low	T0 latch low
&3200044	T0 count high	T0 latch high
&3200048	—	T0 go command
&320004C	—	T0 latch command
&3200050	T1 count low	T1 latch low
&3200054	T1 count high	T1 latch high
&3200058	—	T1 go command
&320005C	—	T1 latch command
&3200060	T2 count low	T2 latch low
&3200064	T2 count high	T2 latch high
&3200068	—	T2 go command
&320006C	—	T2 latch command
&3200070	T3 count low	T3 latch low
&3200074	T3 count high	T3 latch high
&3200078	—	T3 go command
&320007C	—	T3 latch command

Other devices

The devices and peripheral controllers are mapped into memory in these locations:

Address	Type	Bank	IC	Use
&3240000	Slow	4	—	Internal Expansion cards
&3270000	Slow	7	—	External Expansion cards
&32C0000	Med	4	—	Internal Expansion cards
&32D0000	Med	5	HD63463	Hard Disc register write
&32D0008	Med	5	HD63463	Hard Disc DMA read
&32D0020	Med	5	HD63463	Hard Disc register read
&32D0028	Med	5	HD63463	Hard Disc DMA write
&3310000	Fast	1	1772	Floppy disc controller
&3340000	Fast	4	—	Internal Expansion cards
&3350010	Fast	5	HC374	Printer Data
&3350018	Fast	5	HC574	Latch A
&3350040	Fast	5	HC574	Latch B
&33A0000	Sync	2	6854	Econet controller
&33B0000	Sync	3	6551	Serial port controller
&33C0000	Sync	4	—	Internal Expansion cards

Events

Introduction

Events are used by RISC OS to indicate that something specific has occurred. These are typically generated when RISC OS services an interrupt. The SWI OS_GenerateEvent is used to do so. The following events are available:

Number	Event type
0	Output buffer has become empty
1	Input buffer has become full
2	Character has been placed in input buffer
3	End of ADC conversion on a BBC I/O expansion card
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 user remote procedure has been called
9	User has generated an event
10	Mouse buttons have changed state
11	A key has been pressed or released
12	Sound system has reached the start of a bar
13	PC Emulator has generated an event
14	Econet receive has completed
15	Econet transmit has completed
16	Econet operating system remote procedure has been called
17	MIDI system has generated an event

Note that you may generate events yourself, using event number 9, which is reserved for users. You may also get an allocation of an event number from Acorn if you need one – for example, if you are producing an expansion card that generates events.

Enabling and disabling events

Generating events all the time would use a lot of processor time. To avoid this, events are by default disabled. You can enable or disable each event individually.

To avoid problems with several applications using events at the same time, RISC OS keeps a count for each event. This count is increased each time an event is enabled, and decreased when an event is disabled. Thus disabling an event will not stop it being generated if another program still needs the event.

RISC OS sets all event counts to zero at each reset, although some of its system extension modules may need events, and so immediately increment the counts.

Expansion card modules

If the module that is using events has been loaded from an expansion card, it must behave as follows:

- enable the event on all kinds of initialisation
- call OS_Byte 253 on a reset to find out what type it was:
    - if it was a soft reset, enable the event
    - if it was a hard reset or power-on do nothing, as the module will just have been initialised, and so will already have enabled the event
- disable the event on all kinds of finalisation.

Using events

To use event(s), you must first OS_Claim the event vector EventV. See the chapter entitled *Software vectors* for further details. You must then call OS_Byte 14 to enable each of the events you wish to use.

The event routine

When an event occurs, your event routine (that claimed the event vector) is entered. The event number is stored in register R0; other information may be stored in R1 onwards, depending on the event – see below.

The restrictions 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 may only re-enable interrupts if they disable them again before passing on or intercepting the call, and they must ensure that the processing of one event is completed before another is started on. The use of certain operating system calls must be avoided. See the chapter entitled *Interrupts and handling them* for further details.

Finishing with events

When you finish using the events you must first call OS_Byte 13 to disable each event that you originally enabled. You must then OS_Release the event vector EventV.

OS_Byte 13 (SWI &06)

Disables an event

On entry	R0 = 13
	R1 = event number

On exit	R0 preserved R1 = old enable state R2 corrupted
Interrupts	Interrupts are disabled Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant

Use

This call disables an event by decreasing the count of the number of times that event has been enabled. If the count is already zero, it is not altered. 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 13 the same number of times as you use OS_Byte 14.

Related SWIs

OS_Byte 14 (SWI &06), OS_GenerateEvent (SWI &22)

Related vectors

EventV, ByteV

OS_Byte 14 (SWI &06)

Enables an event

On entry	R0 = 14
	R1 = event number

On exit	R0 preserved R1 = old enable state R2 corrupted
Interrupts	Interrupts are disabled Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call enables an event by increasing the count of the number of times that event has been enabled. The previous enable state of the event is returned in R1: R1 = 0 previously disabled R1 > 0 previously enabled When you finish using the vector, you should disable it again by calling OS_Byte 13.
Related SWIs	OS_Byte 13 (SWI &06), OS_GenerateEvent (SWI &22)
Related vectors	EventV, ByteV

OS_GenerateEvent (SWI &22)

Generates an event

On entry	R0 = event number R1… = event parameters
On exit	All registers preserved
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	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 14.
Related SWIs	OS_Byte 13 and 14 (SWI &06)
Related vectors	EventV

Details of events

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

Output buffer empty event

R0 = 0
R1 = buffer number

This event is generated when the last character has just been removed from an output buffer (eg printer buffer, serial port output buffer). See the next chapter entitled *Buffers*.

Input buffer full event

R0 = 1
R1 = buffer number
R2 = byte that could not be inserted into buffer

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 next chapter entitled *Buffers*.

Character input event

R0 = 2
R2 = byte to be inserted into keyboard buffer

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

ADC end conversion event

R0 = 3
R1 = channel that just converted

This event is generated when the analogue-to-digital convertor on the BBC I/O expansion card finishes a conversion. See the documentation supplied with the card.

Vertical sync event

R0 = 4

This event is generated 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 you. In low-resolution modes this will be every fiftieth of a second; in modes requiring a multisync monitor it will be more frequent.

You could use it, for example, to start a timer which will cause a subsequent interrupt. On this interrupt you could change the screen palette, to display more than the usual number of colours on the screen at once.

Interval timer event

R0 = 5

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 3 for details of the interval timer.

Escape event

R0 = 6

This event is generated when either `Esc` is pressed or when an escape condition is received from the RS423 input port. See the chapter entitled *Character input* for a discussion of escape conditions.

RS423 error event

R0 = 7
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 user remote procedure event

R0 = 8
R1 = pointer to argument buffer
R2 = remote procedure call number
R3 = station number
R4 = network number

This event is generated when an Econet user remote procedure call occurs. See the chapter entitled *Econet* for further details.

User event

R0 = 9
R1… = values defined by user

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

Mouse button event

R0 = 10
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

R0 = 11
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
high	0	Esc	‛	Hme	P	G	C	AlR
	1	F1	1	PgU	[	H	V	CtR
	2	F2	2	NL	]	]	B	Lft
	3	F3	3	/	\	K	N	Dwn
	4	F4	4	*	Del	L	M	Rt
	5	F5	5	#	Cpy	:	,	0
	6	F6	6	Tab	PgD	"	.	.
	7	F7	7	Q	7	Ret	/	Ent
	8	F8	8	W	8	4	ShR	
	9	F9	9	E	9	5	Up	
	A	F10	0	R	-	6	1	
	B	F11	-	T	CtL	+	2	
	C	F12	=	Y	A	ShL	3	
	D	Pr	‛	U	S		CapL	
	E	SL	←	I	D	Z	AlL	
	F	Brk	Ins	O	F	X	Spc	

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 Esc at the top left corner.

Note that the keycodes given in this event bear no relationship to any other code you will see. They are not, for example, related to the INKEY numbers described in the chapter *Character input*. They apply to the keyboard supplied on the UK model.

Sound start of bar event

R0 = 12
R1 = 2
R2 = 0

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

The 0 in R2 may change in future versions to give the invocation number of the task causing the event.

PC Emulator event

R0 = 13

This event is claimed by the PC Emulator package.

Econet receive event

R0 = 14
R1 = receive handle
R2 = status of completed operation

This event is generated when an Econet reception completes. The status returned in R2 will always be 9 (Status_Received). See the chapter entitled *Econet* for furhter details.

Econet transmit event

R0 = 15
R1 = transmit handle
R2 = status of completed operation

This event is generated when an Econet transmission completes. The status returned in R2 can have the following values:

0	Transmitted
1	Line jammed
2	Net error
3	Not listening
4	No clock

See the chapter entitled *Econet* for furhter details.

Econet OS remote procedure event

R0 = 16
R1 = pointer to argument buffer
R2 = remote procedure call number
R3 = station number
R4 = network number

This event is generated when an Econet operating system remote procedure call occurs. Current remote procedure call numbers are:

0	Character from Notify
1	Initialise Remote
2	Get View parameters
3	Cause fatal error
4	Character from Remote

See the chapter entitled *Econet* for further details.

MIDI event

R0 = 17
R1 = event code

This event is generated when certain MIDI events occur. The values R1 may have are:

0	A byte has been received when the buffer was previously empty
1	A MIDI error occurred in the background
2	The scheduler queue is to empty, and you can schedule more data.

These events only occur if you have fitted an expansion card with MIDI sockets. See the manual supplied with the card for further details.

Buffers

Introduction

The interrupt system on a RISC OS computer makes extensive use of buffers. These act as temporary holding areas for data after you (or a device) generate it, and before a device (or you) consume it. 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 your program is ready to use it.

Filing system buffers

We are not concerned with filing system buffers in this section. However, these are areas where RISC OS holds whole areas of files in memory to increase the efficiency of file access. The use of file buffers is generally invisible to you; there is no direct way of accessing their contents.

Use of buffers

The buffers we are looking at 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
- purging a buffer (clearing its contents).

All these operations are implemented as OS_Bytes – see below.

The buffer is also purged implicitly when the escape condition is cleared – see the chapter entitled *Character input*.

Details of buffers

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

Number	Use	Size
0	Keyboard	255
1	RS423 (input)	255
2	RS423 (output)	191
3	Printer	1023
4	Sound channel 0	3
5	Sound channel 1	3
6	Sound channel 2	3
7	Sound channel 3	3
8	Speech	3
9	Mouse	63

Buffers 2 to 8 are output buffers. They hold data you generate 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 you are ready to read them.

Buffers 4 to 8

Currently, buffers 4 to 8 are not used. They are provided for compatibility with BBC Micro software. Sound buffering and speech are implemented differently on RISC OS hardware than they were on BBC hardware. These buffers are not considered further.

Data format

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 coordinate low
1	Mouse x coordinate high
2	Mouse y coordinate low
3	Mouse y coordinate high
4	Button state
5	Time of button change, byte 0
6	Time of button change, byte 1
7	Time of button change, 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 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.

OS_Byte calls provided

The OS_Bytes used to control buffers are described below.

They are, in fact, just an interface to the vectored buffer routines described in the chapter entitled *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 number of 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. You would claim the memory for this from the relocatable module area. The module could have its own configuration byte held in CMOS RAM to specify the size of the buffer, which it would claim on initialisation.

OS_Byte 15 (SWI &06)

Flushes all buffers, or the current input buffer

On entry	R0 = 15
	R1 = reason code

On exit	R0 preserved R1, R2 corrupted
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined

Use

This call flushes 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 21.

Related SWIs

OS_Byte 21 (SWI &06)

Related vectors

ByteV

OS_Byte 21 (SWI &06)

Flushes a specified buffer

On entry	R0 = 21
	R1 = buffer number

On exit	R0, R1 preserved R2 corrupted
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call flushes the specified buffer.
Related SWIs	OS_Byte 15 (SWI &06)
Related vectors	ByteV

They are called UpCalls because they are calls that RISC OS makes *up* to an application, rather than calls that the application makes *down* to RISC OS. They occur in the foreground, and are hence different to Events, which occur in the background.

There are a number of different reason codes, each of which is described below: Some are made for information only, others allow the application to take appropriate action (such as to prompt for a missing floppy disc to be inserted in the drive). The caller of the UpCall (normally RISC OS) may then look at any returned state, and decide what action to take next. In many cases it will generate an error if the application has not dealt appropriately with the situation.

Writing code to handle UpCalls

Routines that deal with UpCalls should be viewed as system extensions, and so should only call error-returning SWIs ('X' SWIs).

If a routine installed on the vector does deal with the situation it should intercept the call to the vector, as there is no longer any point informing any other routines or the UpCall handler of the situation. If it cannot deal with the situation it must pass the call on, as another may be able to do so.

OS_Byte 138 (SWI &06)

Inserts a byte into a buffer

On entry	R0 = 138
	R1 = buffer number
	R2 = byte to insert

On exit	R0 - R2 preserved C flag = 0 if character inserted C flag = 1 if buffer was full
nterrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Jse	This call inserts the byte specified in R2 into the buffer identified by R1. If C=1 on exit, the byte was not inserted as there was no room. Inserting bytes 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 a real mouse transition from entering data into the middle of your data. You must do so as quickly as possible to prevent latency in the interrupt system.
Related SWIs	OS_Byte 153 (SWI &06)
Related vectors	ByteV, InsV

To use OS_UpCall 1 or 2, you must either claim UpCallV and install a routine on the vector, or install an UpCall handler. Your routine should:

- prompt you to supply the media with a string built up using:
  - the media type string (passed in R6)
  - the filing system name (obtained by calling XOS_FSControl 33 acting on the value of R1)
  - the media name (passed in R2)

  for example:

  Please insert **disc adfs:Mike** and press Space (Esc to abort)
- give you a way of indicating that you have either supplied the media, or wish to cancel the operation
- intercept the vector with R0 = –1 if you wish to cancel the operation.
- intercept the vector with R0 = 0 if the timeout limit is reached, or if you say you have supplied the media

When you intercept the call to the vector, control passes back to the filing system routine that called OS_UpCall:

- If R0 = –1, then the routine calls OS_UpCall 4; it then returns an error to say that the media was not found.
- If R0 = 0, then the routine checks for you that the media has been changed and the correct one supplied. If so, it calls OS_UpCall 4; otherwise it just calls OS_UpCall 1 or 2 again, after incrementing R4.

The timeout period in R5 is set to a small value for media that can detect when the media has been changed (such as floppy disc drives) and to a large value (typically &FFFFFFFF) for other media. In the former case, this means that RISC OS will automatically detect that new media has been supplied, and check that it is the correct one.

The most common use of OS_UpCall 1 and 2 is to request that a floppy disc is inserted.

Related SWIs

OS_UpCall 4 (SWI &33)

Related vectors

UpCallV

OS_Byte 152 (SWI &06)

Examines the status of a buffer

On entry	R0 = 152
	R1 = buffer number

On exit	R0, R1 preserved R2 = next byte in buffer, or corrupted if buffer was empty C flag = 0 if bytes were in buffer C flag = 1 if buffer was empty
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call returns the status of a specified buffer; the carry flag is set if the buffer is empty. If a byte is available, it is returned in R2 but is not removed from the buffer.
Related SWIs	None
Related vectors	ByteV, RemV

It is made when a program calls one of several SWIs provided by the FileSwitch module:

- reason codes 0 - 9 are caused by calls to OS_File (SWI &08)
- reason codes 257 - 259 are caused by calls to OSFind (SWI &0D)
- reason codes 520 - 521 are caused by calls to OS_FSControl (SWI &29).

You may find it helpful to examine the documentation of the above FileSwitch SWI calls.

The following general points apply:

- all strings are null terminated except where specified
- all object names will already have been expanded by FileSwitch, checked for basic validity, and had filing system prefixes stripped.

Note that if a filename is invalid for a given operation (eg you try to create a file with a wildcarded leafname) FileSwitch will generate an error, and no UpCall will be generated.

The call is used by the desktop filer to maintain its directory displays. It is provided for information only; if you wish to use this UpCall, you must not intercept it, nor must you alter the contents of any of these registers used to pass parameters:

R9 = 0

Saving memory to file

R1 = pointer to filename
R2 = load address
R3 = execution address
R4 = pointer to start of buffer
R5 = pointer to end of buffer
R6 = pointer to special field (or 0)

R9 = 1

Writing catalogue information

R1 = pointer to filename
R2 = load address
R3 = execution address
R5 = attributes
R6 = pointer to special field (or 0)

Communications within RISC OS

Introduction

There are some important SWI calls that RISC OS uses to communicate between different parts of itself, or to communicate with application programs. Because these SWI calls are used by lots of different parts of RISC OS, you will find they are referred to in many different places in the manual. It's therefore important that you know of these SWIs to understand such references. Most of the SWIs belong to modules that are described elsewhere in the manual, so we just cross reference them here.

Service calls

OS_ServiceCall is used to pass a service around modules. Modules can decide whether they wish to provide the service, and if so whether they will then pass the service call on to other modules. A reason code in R1 indicates the type of service. You have already seen two examples of OS_ServiceCall – the reason codes to Claim and Release FIQs.

This call is fully documented in the chapter entitled *Modules*.

Window manager SWIs

The window manager provides various SWIs that enable it to communicate with window based programs (notably Wimp_Poll); and further SWIs so that programs can communicate with and pass data to each other (notably Wimp_Message).

These calls are all fully documented in the chapter entitled *The Window Manager*.

UpCalls

The kernel provides the SWI OS_UpCall, which warns applications of particular situations. It calls the vector UpCallV. To use UpCalls, you must either claim the vector and install a routine on it (see the chapter entitled *Software vectors*), or install an UpCall handler (see the chapter entitled *Program Environment*).

They are called UpCalls because they are calls that RISC OS makes *up* to an application, rather than calls that the application makes *down* to RISC OS. They occur in the foreground, and are hence different to Events, which occur in the background.

There are a number of different reason codes, each of which is described below: Some are made for information only, others allow the application to take appropriate action (such as to prompt for a missing floppy disc to be inserted in the drive). The caller of the UpCall (normally RISC OS) may then look at any returned state, and decide what action to take next. In many cases it will generate an error if the application has not dealt appropriately with the situation.

Writing code to handle UpCalls

Routines that deal with UpCalls should be viewed as system extensions, and so should only call error-returning SWIs ('X' SWIs).

If a routine installed on the vector does deal with the situation it should intercept the call to the vector, as there is no longer any point informing any other routines or the UpCall handler of the situation. If it cannot deal with the situation it must pass the call on, as another may be able to do so.

OS_UpCall 1 and 2 (SWI &33)

Warns your program that a filing media is not present (OS_UpCall 1) or not known (OS_UpCall 2)

On entry

R0 = 1 (Media not present) or 2 (Media not known)
R1 = filing system number (for a list, see the chapter entitled *FileSwitch*)
R2 = pointer to a null-terminated media name string, or –1 if irrelevant
R3 = device number, or –1 if irrelevant
R4 = iteration count for repeated issuing of the call (0 initially)
R5 = minimum timeout period (in centiseconds)
R6 = pointer to a null terminated media type string

On exit

R0 = 0 if media changed, –1 if media no longer required, else preserved
R1 - R6 preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is made by RISC OS filing systems when a program tries to access:

- a filing media that it has previously used but can no longer access (R0 = 1)
- a filing media that it has not previously used (R0 = 2).

It calls the UpCall vector.

To use OS_UpCall 1 or 2, you must either claim UpCallV and install a routine on the vector, or install an UpCall handler. Your routine should:

- prompt you to supply the media with a string built up using:
  - the media type string (passed in R6)
  - the filing system name (obtained by calling XOS_FSControl 33 acting on the value of R1)
  - the media name (passed in R2)

  for example:

  Please insert **disc adfs:Mike** and press Space (Esc to abort)
- give you a way of indicating that you have either supplied the media, or wish to cancel the operation
- intercept the vector with R0 = –1 if you wish to cancel the operation.
- intercept the vector with R0 = 0 if the timeout limit is reached, or if you say you have supplied the media

When you intercept the call to the vector, control passes back to the filing system routine that called OS_UpCall:

- If R0 = –1, then the routine calls OS_UpCall 4; it then returns an error to say that the media was not found.
- If R0 = 0, then the routine checks for you that the media has been changed and the correct one supplied. If so, it calls OS_UpCall 4; otherwise it just calls OS_UpCall 1 or 2 again, after incrementing R4.

The timeout period in R5 is set to a small value for media that can detect when the media has been changed (such as floppy disc drives) and to a large value (typically &FFFFFFFF) for other media. In the former case, this means that RISC OS will automatically detect that new media has been supplied, and check that it is the correct one.

The most common use of OS_UpCall 1 and 2 is to request that a floppy disc is inserted.

Related SWIs

OS_UpCall 4 (SWI &33)

Related vectors

UpCallV

OS_UpCall 3 (SWI &33)

Warns your program that a file is being modified

On entry	R0 = 3 (Modifying file)
	R1 - R7 vary, depending on the value of R9
	R8 = filing system information word
	R9 = reason code

On exit	All registers preserved
Interrupts	Interrupt status is unaltered Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call warns your program that a file is being modified. The reason code in R9 tells you how:

R9	**Meaning**
0	Saving memory to file
1	Writing catalogue information
2	Writing load address only
3	Writing execution address only
4	Writing attributes only
6	Deleting file
7	Creating empty file
8	Creating directory
257	Creating and opening for update
259	Closing file
520	Renaming file
521	Setting attributes

It is made when a program calls one of several SWIs provided by the FileSwitch module:

- reason codes 0 - 9 are caused by calls to OS_File (SWI &08)
- reason codes 257 - 259 are caused by calls to OSFind (SWI &0D)
- reason codes 520 - 521 are caused by calls to OS_FSControl (SWI &29).

You may find it helpful to examine the documentation of the above FileSwitch SWI calls.

The following general points apply:

- all strings are null terminated except where specified
- all object names will already have been expanded by FileSwitch, checked for basic validity, and had filing system prefixes stripped.

Note that if a filename is invalid for a given operation (eg you try to create a file with a wildcarded leafname) FileSwitch will generate an error, and no UpCall will be generated.

The call is used by the desktop filer to maintain its directory displays. It is provided for information only; if you wish to use this UpCall, you must not intercept it, nor must you alter the contents of any of these registers used to pass parameters:

R9 = 0

Saving memory to file

R1 = pointer to filename
R2 = load address
R3 = execution address
R4 = pointer to start of buffer
R5 = pointer to end of buffer
R6 = pointer to special field (or 0)

R9 = 1

Writing catalogue information

R1 = pointer to filename
R2 = load address
R3 = execution address
R5 = attributes
R6 = pointer to special field (or 0)

R9 = 2	Writing load address only R1 = pointer to filename R5 = pointer to end of buffer R6 = pointer to special field (or 0)

R9 = 3	Writing execution address only
	
	R1 = pointer to filename
	R3 = execution address
	R6 = pointer to special field (or 0)

R9 = 4	Writing attributes only R1 = pointer to object name R5 = attributes R6 = pointer to special field (or 0)
R9 = 6	Deleting file R1 = pointer to object name R6 = pointer to special field (or 0)

R9 = 7	Creating empty file
	
	R1 = pointer to filename
	R2 = load address
	R3 = execution address
	R4 = start address
	R5 = end address
	R6 = pointer to special field (or 0)

R9 = 8	Creating directory R1 = pointer to directory name R2 = load address (to be used as timestamp) R3 = execution address (to be used as timestamp) R6 = pointer to special field (or 0)
R9 = 257	Creating and opening for update R1 = pointer to filename R2 = external handle file will be given (if successfully opened) R6 = pointer to special field (or 0)

R9 = 259	Closing file R1 = external handle
R9 = 520	Renaming file R1 = pointer to current object name R2 = pointer to desired object name R3 = execution address R6 = pointer to current special field (or 0) R7 = pointer to desired special field (or 0)
R9 = 521	Setting attributes R1 = pointer to object name R2 = pointer to attribute string (control character terminated)
Related SWIs	None
Related vectors	UpCallV

OS_UpCall 4 (SWI &33)

Informs your program that a missing filing media has been supplied, or that an operation involving one has been cancelled

On entry	R0 = 4 (Media search end)
On exit	R0 preserved
Interrupts	Interrupt status is unaltered Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call is made by RISC OS to inform your program that a missing filing media has been supplied, or that an operation involving one has been cancelled. It is always preceded by call(s) of OS_UpCall 1 or OS_UpCall 2. It calls the UpCall vector. To use OS_UpCall 4, you must either claim UpCallV and install a routine on the vector, or install an UpCall handler. This call is typically used to remove error messages displayed when OS_UpCall 1 or 2 was first generated.
Related SWIs	OS_UpCall 1 and 2 (SWI &33)
Related vectors	UpCallV

OS_UpCall 256 (SWI &33)

Warns your program that a new application is going to be started

On entry	R0 = 256 (New application)
	R2 = proposed Currently Active Object pointer

On exit	R0 = 0 to stop application, else R0 is preserved
Interrupts	Interrupt status is unaltered Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call is made just before a new application is going to be started – for example due to a *RUN or module command. It calls the UpCall vector. To use OS_UpCall 256, you must either claim UpCallV and install a routine on the vector, or install an UpCall handler. One reason to use this call is so that an application can tidy up after itself before a new one starts, eg removing routines from vectors. For more details, see the chapter entitled *The Program Environment*. Another reason to use this UpCall is to prevent an application from starting. If you don't want the application to start, your routine should set R0 to 0, and intercept the call to the vector. This will cause the error `Unable to start application` to be given. Otherwise, you must pass the call on with all registers preserved.
Related SWIs	None
Related vectors	UpCallV

OS_UpCall 257 (SWI &33)

Informs your program that RISC OS would like to move memory

On entry

R0 = 257 (Moving memory)
R1 = amount that application space is going to change by

On exit

R0 = 0 to permit memory move, else R0 is preserved
R1 is preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is made just before OS_ChangeDynamicArea tries to move memory. The call is only made if the currently active object is in the application space. It calls the UpCall vector. By default (if you do not claim the vector) the memory is not moved.

To allow the memory to be moved, you must either claim UpCallV and install a routine on the vector, or install an UpCall handler. Your routine must shuffle your application's workspace so that the memory move can go ahead. It must then set R0 = 0, and pass on the call to the vector.

Related SWIs

None

Related vectors

UpCallV

Part 2 - The kernel

Character Output

Introduction

The Character Output system can send characters to the computers' output devices. They can be any or all of the following:

- the VDU
- the serial port
- a file on any filing system
- the currently selected printer

The Character Output system gives full control of the operation of each of these devices. Since they all have different characteristics, they must be controlled in different ways.

Character Output provides a means of directing characters to the device(s) that are required. It is like a train shunting yard that can send characters, like trains, to the right destination. It can also hold them, waiting until the destination is free to take them.

Overview

The Character Output system can be divided by an imaginary horizontal line. Above it is the part independent of the device(s) that the characters will end up at. Below the line is the control of each of the devices.

Terminology used

- A device is the hardware that is used to send characters to some external form, such as shapes on a VDU or voltages on a serial line or onto a floppy disk and so on.
- A port is like a device, though it really refers more to the actual connection to the outside.
- A device driver is the low level code that operates a device.
- A stream is like its normal usage. It is a connection between a program and a device. Streams can also go from one program to many devices.

Back-doors

Normally, a program will go through the stream system to access output devices. However, 'back-doors' are provided to allow directly writing to a given device. A major reason for wanting to do this is speed, since the stream system necessarily takes time. Another is that this back-door approach gives much more direct control of the device and more immediate feedback on problems. A modem driving program, for example, needs to be able to react quickly to information on the serial line.

Device independence

Device independence means that any program using the stream system doesn't have to know the destination of the characters it is outputting. Most programs don't, since it will not affect their actions. If they do need to, then back-doors are available.

OS_WriteC

The core of the stream system is the SWI OS_WriteC which outputs a single character. It looks at which of the devices have been enabled and sends a copy of the character to each of them. It is in turn called by many other SWIs, printing a string for example. Characters from these other SWIs stream into OS_WriteC and from there out to the correct device.

Buffers

A program running in RISC OS works at one rate, while the hardware devices all work at different rates. This is called asynchronous operation, since the two are not synchronised. To solve this problem, buffers are used. A buffer is simply an area of memory that has been set aside to temporarily hold data. RISC OS provides buffering for all the devices used by the stream system. A program will write into a buffer, while interrupts asynchronously read it out. If a buffer became full, then RISC OS would wait until it had emptied somewhat, then continue, without the calling program ever being aware it had happened.

Devices

OS_WriteC can be setup to send to one or many of the following list of devices:

- the printer stream
- the serial driver
- the spool (filing system) driver
- the VDU driver

The control of which devices are enabled at any time is very simple and can be changed as frequently or infrequently as desired.

These are briefly summarised below, and described in depth in later sections.

Printer stream

There are several ways in which the printer stream may be directed. Unlike the high level output streams previously discussed, where several devices may be used at once, only one printer device may be active at any one time. The printer stream is, in effect, a subpart of the full stream system.

Like the stream system, the printer stream has a number of devices it can use. The ones available are:

- Printer sink
- Centronics parallel
- serial port
- network printer
- user printer driver

The printer sink is a special case. Unlike the other drivers, which operate some hardware, the printer sink is a null printer device. This simply absorbs any characters sent to it. For example, it is a device that can be used when you don't want any form of printer output with an application that uses the printer.

The Centronics parallel device allows printing on any standard parallel printer. This includes virtually all of the low cost printers sold.

The RS423 serial device can be connected to any serial printer. RS423 is like the more usual RS232 serial standard, but is better whilst still being compatible with any RS232 device.

The network printer is the one that is accessed remotely across a network. See the chapter entitled *NetPrint* for details of this.

Finally, the user printer driver allows programmers to write a driver to support a device not listed here.

Note that this chapter concerns itself only with the character print routines. See the chapter entitled *Printer Drivers* for information on the drivers that must be used for any graphical printing.

Serial output device

The device driver software takes characters from the stream system and puts them into the serial hardware, manipulating it to send them off.

The serial hardware itself changes the character into a series of voltage changes on its connection with the outside. These voltages and other control lines work together to communicate with another serial port on another machine. The baud rate of a serial port is the number of bits per second that it is sending or receiving. Under RISC OS, these rates can be controlled independently.

In this chapter, the output-specific and general calls to this device are covered. In the chapter entitled *Character Input*, the input-specific calls are described.

Spool device

In RISC OS, you can *spool* characters to a file on a filing system as if it were a sequential device. The term itself is an archaic one that has passed down from early mainframe computers.

It is very easy to use a spool file. There is a command to start spooling output to a named file, and another to stop spooling and close the file. Also, you can change the file you are spooling to at any time, without having to close and re-open it.

VDU device

The VDU device driver will put any characters or graphics onto the screen. Some characters are displayed directly, while others are interpreted as graphics commands. This chapter contains details of the interface to the VDU system, but for a detailed description of the VDU system, refer to the chapter entitled *VDU drivers*.

Technical Details

Device independence

The core of the output stream is the SWI OS_WriteC. This is called via WrchV, the Write Character vector. Note that if this vector is ever replaced then all of the other routines that use it will also be redirected. OS_WriteC is called by many, many routines; in this chapter OS_WriteS, OS_Write0, OS_WriteN, OS_NewLine, OS_PrettyPrint and OS_WriteI.

OS_Byte 3 controls which devices characters get sent to. It sets a byte in which each bit represents a different output device state. Some of these bits enable whether a device gets characters or not. There are complications however, which are described fully in the following sections.

Printer stream

The printer stream can be enabled by OS_Byte 3 or using VDU codes. The selection of the printer is done by OS_Byte 5. The printer can be made to ignore a specific character by using OS_Byte 6.

OS_Byte 3

Three bits in the byte sent to OS_Byte 3 to select output streams 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.

VDU printer control

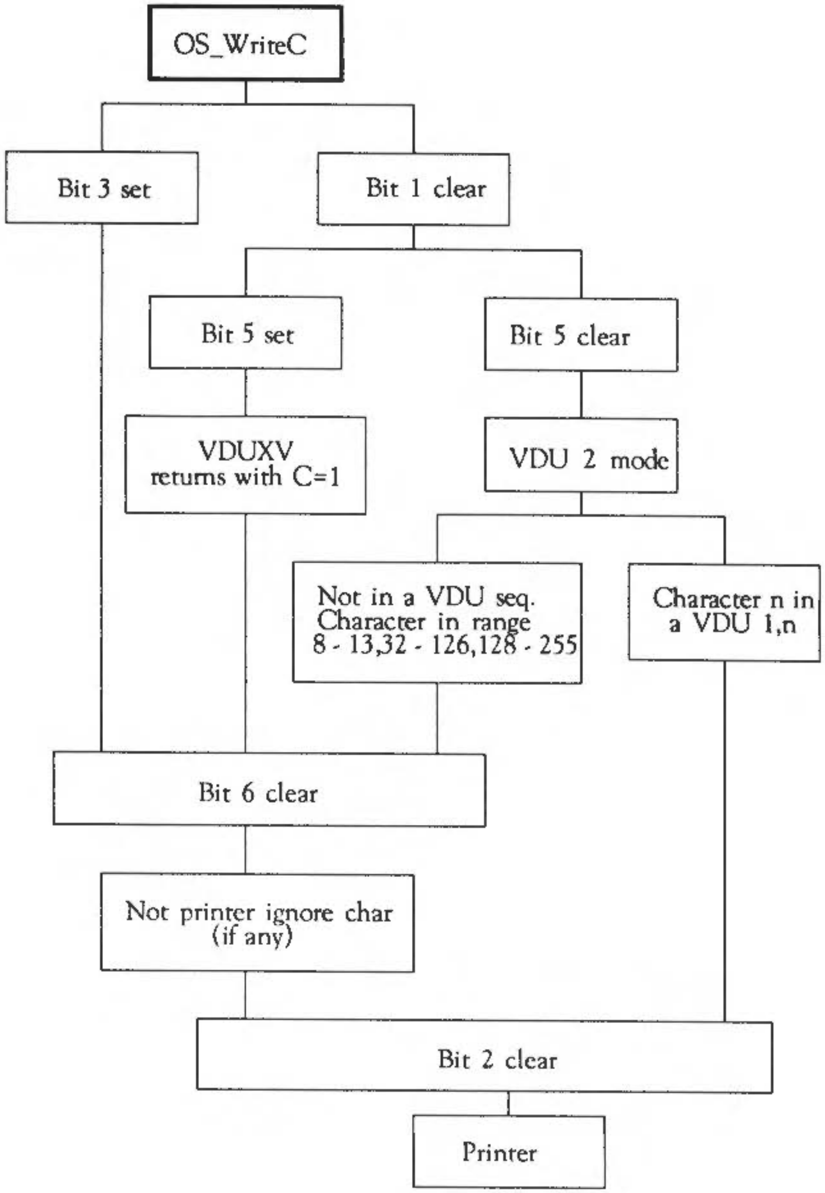
The most common way of controlling the printer is through the VDU driver. If the VDU stream is enabled (bit 1 of the output stream's byte is clear), then sending the code ASCII 2 (Ctrl-B) to OS_WriteC enables the VDU printer stream. Once this is done, all printable characters and some control characters 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 stream, and the printer ignore character.

If either bit 6 or bit 2 of the streams byte is set, then the VDU printer stream 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.

More details of the VDU printer stream control codes are given in the chapter entitled *VDU driver*.

The flow of control is summarised by the diagram on the following page:


OS_WriteC
Bit 3 set
Bit 1 clear
Bit 5 set
Bit 5 clear
VDUXV returns with C=1
VDU 2 mode
Not in a VDU seq. Character in range 8 - 13,32 - 126,128 - 255
Character n in a VDU 1,n
Bit 6 clear
Not printer ignore char (if any)
Bit 2 clear
Printer

OS_Byte 5

Regardless of how a character gets to the printer stream, it is then sent to the current printer device . This is set by OS_Byte 5. It is passed a byte which can select one of 256 potential drivers, 4 of which are supplied with RISC OS.

- printer sink
- parallel
- serial
- network

When an OS_Byte 5 is used, the new destination streams come 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, the calling task may appear to hang until the current printer buffer's contents are cleared. This may be forced by generating an escape condition.

The default printer device is stored in CMOS RAM and is set by `*Configure~Print`.

OS_Byte 245

OS_Byte 245 (SWI &F5) may be used to read the current printer type, 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, so may be used to read the printer type from within an interrupt routine.

Ignore character

The printer ignore character is one which is suppressed from the printer stream, unless it got there via the VDU printer stream and was preceded by ASCII 1 (Ctrl~A). The character can be set and read using OS_Byte 246. For compatibility with older Acorn operating systems, OS_Byte 6 can also set it and OS_Byte 245 can read it.

`*Ignore <number>` can be used to set it from the CLI. `*Configure Ignore <number>` will set it permanently in CMOS RAM. The default value is 10, an ASCII linefeed.

No ignore

There may be no printer ignore character, in which case all characters are sent. This is called the NOIGNORE state and can be set with OS_Byte 182.

`*Ignore` with no parameter has the same effect from the CLI. `*Configure Ignore` will set it permanently in CMOS RAM.s

Serial Device

The serial device driver provides facilities to send and receive a byte, control the handshake lines and alter the protocol of the data. RISC OS provides a number of SWIs that allow access to these facilities.

Sending a byte

There are two fundamental ways of communicating with the serial port.

- OS_Bytes 3 and 5 can be used to select it as an output stream. OS_WriteC and the SWIs that use it would be used to write to its buffer, with RISC OS handling buffer full conditions and so on.
- The OS_SerialOp SWI contains routines to access the serial device driver directly. This SWI is like OS_Byte in that it contains a number of operations, determined by the reason code passed in R0. The advantages of using this approach are the speed of not going through several routines in the stream system and no possibility of confusion about where the data is going.

Controlling the port

Sending characters is, of course, only the start. The control of the state of the serial device can likewise be handled in two ways. The OS_Byte serial port commands are in RISC OS mainly for compatibility with earlier Acorn operating systems. It is recommended that the OS_SerialOp commands are used in preference to the OS_Bytes because they are more complete and consistent.

Note that the serial device's input and output sides may be controlled independently. For example, you can transmit at a different baud rate from the one which is being used to receive.

Control commands

Here is a summary of the OS_Byte and OS_SerialOp commands:

- OS_Byte 156 is a bit mask that reads and writes various state information.
- OS_Byte 192 reads the above state byte.
- OS_Byte 8 sets the transmit baud rate (7 handles the receive rate).
- OS_Byte 191 reads and writes the busy flag (obsolete BBC usage)
- OS_Byte 242 reads both baud rates.
- OS_SerialOp 0 reads and writes the handshaking status.
- OS_SerialOp 1 reads and writes the data format.

- OS_SerialOp 2 sends a break.
- OS_SerialOp 6 reads and writes the transmit baud rate (5 for receive).

OS_Byte 3 and 5

When bit 0 of the OS_Byte 3 streams byte is set, characters sent to OS_WriteC are passed to the serial output stream. In particular, they are inserted into the serial output buffer (buffer number 2), where they remain until removed by the interrupt routine dealing with serial transmission.

Note that if the serial port is selected as the printer by OS_Byte 5, and the serial port is enabled by setting bit 0 of the stream's byte with OS_Byte 3, then the character is inserted into both buffers. This means that eventually the character is printed twice, first from the serial output buffer and then from the printer buffer. To solve this problem, make the printer another device type, such as the printer sink, which allows data sent to the printer to be ignored.

Serial output buffer

If the output buffer is already full and there is nothing communicating with the serial port, when you insert another character the machine temporarily halts while it waits for a character to be removed to make space for the new character. An escape condition abandons this wait.

Handshaking and protocol

When trying to get communications working with an external device using the serial device, there are several important factors to remember:

- The receiver must be electrically compatible with RS423 or RS232.
- The handshaking lines must be connected between the sender and receiver in exactly the right way.
- The sender must match baud rates with the receiver.
- They must also match the transmission protocol. Each byte sent is packaged up in some variation of the following sequence:

1. A start bit synchronises the receiver with the sender.
2. The number of bits of actual data sent is variable from 5 to 8.
3. There can be an optional parity bit, which is used to check that no errors have taken place during transmission.
4. It ends with a stop bit, either 1, 1.5 or 2 bits long.

Note that the default setup of the serial protocol (configured in CMOS RAM) is different from some earlier Acorn machines. For example, the setup for RISC OS machines is the same as the Master series (8 data bits, no parity, 2 stop bits), but different from the original BBC series (8 data bits, no parity, 1 stop bit).

Serial line names

Coming out of the serial connector are many lines. This is a list of their names and common abbreviations:

- data receive (RxD)
- data transmit (TxD)
- ground (0V)
- request to send (RTS)
- confirm to send (CTS)
- data carrier detect (DCD)
- data terminal ready (DTR)
- data set ready (DSR)

Refer to the documentation accompanying your particular communications device for information on how to wire these lines correctly with the serial port. For further information, contact Acorn Customer Support.

Spool device

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.

Opening and closing

There are two ways of opening and closing a spool file. The simplest is to use the CLI commands `*Spool <pathname>` or `*SpoolOn <pathname>` to start output going into the named file.

To stop spooling and close the file, a `*Spool` or `*SpoolOn` command with no parameters must be issued, or you can stop it directly by using OS_Byte 199 documented below.

OS_Byte 3

The spool file stream can be temporarily disabled by setting bit 4 of the streams byte in OS_Byte 3. This does not close the file, but prevents OS_WriteC from trying to send the character to file.

OS_Byte 199

OS_Byte 199 (SWI &C7) provides direct control over the spool file, without the necessity of using the CLI. 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. You will only need to use this command for sophisticated programs that, say, keep swapping between several spool files.

VDU device

The VDU driver will display characters and graphics on the screen. The value of the character sent determines its effect. Below is a list of the meanings of different characters. Note that in Teletext modes, a different set is in use.

• 0 - 31	– VDU commands (graphics and control)
• 32 - 126	– ASCII characters
• 127	– Delete
• 128 - 159	– User definable characters
• 160 - 255	– ISO international characters

Note that if defining characters in the range 128 - 159 under the Desktop, you should always read the current definition of the character first using OS_Word 10 and then redefine it for the duration of the redraw. Always ensure that the character definition is restored (*not* set to the default using *FX 25) before calling XWimp_Poll again.

Disabling VDU driver

If an OS_Byte 3 with bit 1 set is sent, then the VDU driver is disabled. This prevents all output from appearing on the screen. Also, as control codes will not be acted on, it disables the VDU printer stream, described in an earlier section.

Disabling the VDU, by setting this bit, is independent of the ASCII 21 (Ctrl-U), which will disable the VDU drivers. The main difference is that the VDU printer stream will still work, if already enabled by ASCII 2 (Ctrl-B), after an ASCII 21.

VDUXV

VDUXV is the VDU extension vector. When an OS_Byte 3 with bit 1 clear (VDU~enabled) and bit 5 set (VDUXV enabled) is issued, 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, for example, uses this facility.

The character sent to VDUXV can be sent to the printer stream by setting the carry flag on return from the vector.

See the chapter entitled *Software vectors* for more details on installing a routine on this vector.

Direct Control

OS_Plot can be used to write to the VDU directly rather than going through the stream system. It is consequently faster. It is described in the chapter entitled *VDU drivers*.

OS_WriteC (SWI &00)

Write a character to all of the active output streams

On entry	R0 = character to write
On exit	R0 = preserved
Interrupts	Interrupts are enabled Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant

Use

This call sends the byte in R0 to all of the active output streams. This is called as a low level writer by several other routines.

OS_WriteC calls the Write Character vector WrchV, the default action of which is to send the character to all active output streams. If this vector is replaced, using OS_Claim, then all of the SWIs that use this vector will be funnelled into the replacement routine.

All the routines that call OS_WriteC may not actually call OS_WriteC or even WrchV unless there is some pressing reason to do so. For example, if WrchV is being intercepted by someone else as well as the defualt ROM routine, if a spool file is active or the printer is active etc.

Related SWIs

OS_WriteS (SWI &01), OS_Write0 (SWI &02), OS_NewLine (SWI &03), OS_PrettyPrint (SWI &44), OS_WriteN (SWI &46), OS_WriteI (SWI &100 - SWI &1FF), OS_Byte 3 (SWI &06)

Related vectors

WrchV

OS_WriteS (SWI &01)

Write the following string to all of the active output streams

On entry	—
On exit	—
Interrupts	Interrupts are enabled Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call sends the string that immediately follows the SWI instruction to all of the active output streams. It uses OS_WriteC directly a character at a time. The string is terminated by a null. This SWI alters its return address so that execution continues at the word after the end of the string. Consequently you must not conditionally execute this SWI.
Related SWIs	OS_WriteC (SWI &00)
Related vectors	WrchV

OS_Write0 (SWI &02)

Write a indirect string to all of the active output streams

On entry	R0 = pointer to null-terminated string to write
On exit	R0 = points to the byte after the null byte
Interrupts	Interrupts are enabled Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call sends the string pointed to by R0 to all of the active output streams. It uses OS_WriteC directly a character at a time.
Related SWIs	OS_WriteC (SWI &00)
Related vectors	WrchV

OS_NewLine (SWI &03)

Write a line feed followed by a carriage return to all of the active output streams.

On entry	—
On exit	—
Interrupts	Interrupts are enabled Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant

Use

This call uses OS_WriteC. It is equivalent to two calls to OS_WriteI. For example:

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

This can now be replaced by a single call:

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

Related SWIs	OS_WriteC (SWI &00), OS_WriteI (SWI &100 - SWI &1FF)
Related vectors	WrchV

OS_Byte 3 (SWI &06)

Specify output streams

On entry

R0 = 3 (reason code)
R1 = determines the output stream(s)

On exit

R0 = preserved
R1 = previous stream specification
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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 serial driver
1	Disables VDU driver
2	Disables VDU printer stream
3	Enables printer (independently of the VDU)
4	Disables spooled output
5	Calls VDUXV instead of VDU driver (see the chapter on VDU)
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 stream is enabled, and the spool stream is enabled.

Details of how bits 1, 2, 3 and 6 interact is described in the *Technical Details* section of this chapter.

This command can also be performed by `*FX 3,<stream byte>`

Related SWIs	OS_Byte 236 (SWI &06)
Related vectors	ByteV, VDUXV, WrchV

OS_Byte 5 (SWI &06)

Write printer driver type

On entry

R0 = 5 (reason code)
R1 = driver type

On exit

R0 = preserved
R1 = previous driver type
R2 = corrupted

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The driver is determined as follows:

Value	Type
0	Printer sink
1	Parallel (Centronics) printer driver
2	Serial output
3 - 255	Files in system variables PrinterType$n (eg the NetPrint module sets up PrinterType$4)

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 serial port is selected as the printer, and the serial 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 serial output buffer), 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 this OS_Byte is issued, or the corresponding `*FX` command, the machine may appear to hang until the current printer buffer's contents are cleared. (This may be forced to happen by acknowledging an escape condition from the foreground provided that the escape side effects are enabled.)

This command can also be performed by `*FX 5,<driver type>`

Related SWIs

OS_Byte 8 (SWI &06), OS_Byte 245 (SWI &06)

Related vectors

ByteV

OS_Byte 6 (SWI &06)

Write printer ignore character

On entry	R0 = 6 (reason code) R1 = ASCII value of ignore character

On exit	R0 = preserved
	R1 = previous ignore character
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	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. This command can also be performed by `*FX 6,<ignore character>`
Related SWIs	OS_Byte 246 (SWI &06), OS_Byte 182 (SWI &06)
Related vectors	ByteV

OS_Byte 8 (SWI &06)

Write serial port transmit rate

On entry

R0 = 8 (reason code)
R1 = baud rate code

On exit

R0 = preserved
R1 = corrupted
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the serial 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
9	50
10	110
11	134.5
12	600
13	1800
14	3600
15	7200

The settings from 0 to 8 are in an order compatible with earlier operating systems. The other speeds from 9 to 15 provide all the other standard baud rates.

The default rate is that set by `*Configure Baud`.

This command can also be performed by `*FX 8,<baud rate>`

Related SWIs

OS_Byte 7 (SWI &06)

Related vectors

ByteV

OS_Byte 156 (SWI &06)

Read/write serial communications state

On entry

R0 = 156 (SWI &9C) (reason code)
R1 = 0 or new value
R2 = 255 or 0

On exit

R0 = preserved
R1 = value before being overwritten
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1. ie ((value AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call accesses the control byte of the serial port. In addition to updating the status byte in RAM, it also updates the hardware register which controls the serial port characteristics.

The call enables the current settings of the transmitter, receiver, interrupts and the serial handshake line 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 low, transmit break level on transmit data, transmit interrupt disabled

The above bits should not be modified as they are controlled by the OS. Use the OS_SerialOp SWIs instead to control transmission.

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 192. The write command can also be performed by `*FX 156,<new value>`.

OS_SerialOps 0 and 1 provide all of these facilities and more, with the exception of the interrupt control bit. The receive interrupt/control bit can be set/cleared via OS_Byte 2. You should not change the RTS/transmit IRQ bits; RISC OS handles this function.

This call is provided for compatibility only and should not be used. In all cases you should use OS_SerialOp to provide these functions.

Related SWIs

OS_SerialOp (SWI &57) OS_Byte 192 (SWI &06)

Related vectors

ByteV

OS_Byte 182 (SWI &06)

Read/write NOIGNORE state

On entry

R0 = 182 (&B6) (reason code)
R1 = 0 to read or new state to write
R2 = 255 to read or 0 to write

On exit

R0 = preserved
R1 = state before being overwritten
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The state stored is changed by being masked with R2 and then exclusive ORd with R1. ie ((state AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call allows reading the current NOIGNORE state or changing it to a new value.

If the value read or written is >=&80 (i.e 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 write command can also be performed by `*FX 182,<new state>`

Related SWIs

OS_Byte 6 (SWI &06), OS_Byte 246 (SWI &06)

Related vectors

ByteV

OS_Byte 191 (SWI &06)

Read/write serial busy flag

On entry	R0 = 191 (&BF) (reason code)
	R1 = 0 or new value
	R2 = 255 or 0

On exit	R0 = preserved R1 = state before being overwritten R2 = value of serial port control byte (see OS_Byte 192)
Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call is provided for compatibility reasons only; the cassette interface and RS423 serial port shared the same hardware on the BBC/Master 128 machines. It performs no useful function under RISC OS.
Related SWIs	None
Related vectors	ByteV

OS_Byte 192 (SWI &06)

Read serial communications state

On entry	R0 = 192 (&C0) (reason code)
	R1 = 0
	R2 = 255

On exit	R0 = preserved R1 = value of communications state R2 = value of flash counter (see OS_Byte 193)
Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call reads the control byte of the serial port. It is equivalent to a read operation with OS_Byte 156. This call should not be used to write the value back, as to do so would make the RISC OS copy of the register inconsistent with the actual register in the serial hardware.
Related SWIs	OS_SerialOp (SWI &57) OS_Byte 156 (SWI &06)
Related vectors	ByteV

OS_Byte 199 (SWI &06)

Read/write spool file handle

On entry	R0 = 199 (&C7) (reason code)
	R1 = 0 to read or new handle to write
	R2 = 255 to read or 0 to write

On exit	R0 = preserved
	R1 = handle before being overwritten
	R2 = corrupted

'nterrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
'Jse	The handle stored is changed by being masked with R2 and then exclusive ORd with R1. ie ((handle AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits. This call is used to set where spooled data is sent. If its value is zero or if spooling is disabled by OS_Byte 3, then no data is sent. A handle must be correctly returned from a call to OS_Find. If this handle is then passed to this routine then data will be sent to that file if spooling is enabled. The write command can also be performed by `*FX 199,<handle>`
Related SWIs	OS_Byte 3 (SWI &06), OS_Find (SWI &0D)
Related vectors	ByteV

OS_Byte 236 (SWI &06)

Read/write character destination status

On entry

R0 = 236 (&EC) (reason code)
R1 = 0 when reading or new status when writing
R2 = 255 to read or 0 to write

On exit

R0 = preserved
R1 = status before being overwritten
R2 = cursor key status (see OS_Byte 237)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The status stored is changed by being masked with R2 and then exclusive ORd with R1. ie ((status AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call reads and writes the output streams value. This can also be written by OS_Byte 3. See OS_Byte 3 for a list of the bit values.

The write command can also be performed by `*FX 236,<status>`

Related SWIs

OS_Byte 3 (SWI &06)

Related vectors

ByteV

OS_Byte 242 (SWI &06)

Read serial baud rates

On entry

R0 = 242 (&F2) (reason code)
R1 = 0
R2 = 255

On exit

R0 = preserved
R1 = baud rates
R2 = timer switch state (see OS_Byte 243)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

R1 returns an encoded value which gives the baud rate for serial receive and transmit. Originally, in the BBC/Master operating systems, only eight baud rates were available. These could be encoded in three bits each for receive and transmit. Under RISC OS, 15 are available, which requires four bits to encode. For compatibility with this earlier format, the layout of this byte looks unusual:

Bit	Meaning
0	Transmit bit 0
1	Transmit bit 1
2	Transmit bit 2
3	Receive bit 0
4	Receive bit 1
5	Receive bit 2
6	Receive bit 3
7	Transmit bit 3

These four bit groups are encoded with baud rates. Note that this order is not the same as the order used by any other baud rate setting SWI. This order is based on the original hardware:

Value	Baud Rate
0	19200
1	1200
2	4800
3	150
4	9600
5	300
6	2400
7	75
8	7200
9	134.5
10	1800
11	50
12	3600
13	110
14	600
15	undefined

The value stored must not be changed by making R1 and R2 other than the values stated above.

This call is provided for backwards compatibility with the BBC and Master operating systems. You should in preference use OS_SerialOps 5 and 6 to read and write baud rates.

Related SWIs

OS_Byte 7 (SWI &06), OS_Byte 8 (SWI &06), OS_SerialOp (SWI &57)

Related vectors

ByteV

OS_Byte 245 (SWI &06)

Read printer driver type

On entry	R0 = 245 (&F5) (reason code)
	R1 = 0
	R2 = 255

On exit	R0 = preserved R1 = value before being overwritten R2 = value printer ignore character (see OS_Byte 246)
Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined

Use

The value stored must not be changed by making R1 and R2 other than the values stated above. Use OS_Byte 5 instead to write.

This call will return values in R1 in the following range:

Value	Type
0	Printer sink
1	Parallel (Centronics) printer driver
2	Serial output
3 - 255	Files in system variables PrinterType$n (eg the NetPrint module uses PrinterType$4)

This call 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.

Related SWIs

OS_Byte 5 (SWI &06)

Related vectors

ByteV

OS_Byte 246 (SWI &06)

Read/write printer ignore character

On entry

R0 = 246 (&F6) (reason code)
R1 = 0 to read or new ASCII value to write
R2 = 255 to read or 0 to write

On exit

R0 = preserved
R1 = value before being overwritten
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1. ie ((value AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call allows reading the current state of the printer ignore character or changing it to a new value.

The write command can also be performed by `*FX 246,<value>`

Related SWIs

OS_Byte 6 (SWI &06), OS_Byte 182 (SWI &06)

Related vectors

ByteV

OS_PrettyPrint (SWI &44)

Write an indirect string with some formatting to all of the active output streams

On entry

R0 = pointer to null-terminated string to write
R1 = pointer to dictionary (0 means use the internal RISC OS dictionary)
R2 = pointer to null-terminated special string

On exit

R0 = preserved
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call acts like OS_Write0, with several differences:

- Several characters have special meanings to OS_PrettyPrint.
- It will break a line at a SPACE (ACSII 32) if the next word will not fit on the line; it will not do this at hard spaces.
- Compacted text is handled.

The following characters in the string have special meanings:

- CR (ASCII 13) causes a newline to be generated.
- TAB (ASCII 9) causes a tabulation to the next multiple of eight columns.
- SPACE (ASCII 31) is a hard space.
- ESC (ASCII 27) indicates that a dictionary entry should be substituted.

Compacted text uses an escape character in the print string to indicate a dictionary entry. It is followed immediately by a byte which is the dictionary entry number. If this byte is in the range 1 to 255, then the appropriate string in the dictionary is substituted. If it is 0, then the special string pointed to by R2 on entry is substituted. (This is used in particular by the *Help command.)

The format of a dictionary is a linear list of entries, these entries can recursively refer to other dictionary entries; each entry is a length byte followed by a null-terminated string. This means that a dictionary does not have to have 255 entries. It can be ended at any point with a zero length entry.

The contents of the RISC OS dictionary is summarised below:

Token	**String**
0	<string pointed to by R2>
1	`"Syntax: *"`<string pointed to by R2>
2	`" the "`
3	`"director"`
4	`"filing system"`
5	`"current"`
6	`" to a variable. Other types of value can be assigned with *"`
7	`"file"`
8	`"default "`
9	`"tion"`
10	`"*Configure "`
11	`"name"`
12	`" server"`
13	`"number"`
14	`"Syntax: *"`<string pointed to by R2>`" <"`
15	`" one or more files that match the given wildcard"`
16	`" and "`
17	`"relocatable module"`
18	<CR>`"C(onfirm)"`<TAB>`"Prompt for confirmation of each "`
19	`"sets the "`
20	`"Syntax: *"`<string pointed to by R2>`" [<disc spec.>]"`
21	`")"`<CR>`"V(erbose)"`<TAB>`"Print information on each file "`
23	`"spriteLandscape [<XScale> [<YScale> [<Margin> [<Threshold>]]]]]"`
24	`" is used to print a hard copy of the screen on EPSON-"`
25	`"."`<CR>`"Options: (use ~ to force off, eg. ~"`
26	`"printe"`

27	`"Syntax: *"`<string pointed to by R2>`" <filename>"`
28	`"select"`
29	`"xpression"`
30	`"Syntax: *"`<string pointed to by R2>`" ["`
31	`"sprite"`
32	`" displays"`
33	`"free space"`
34	`" (off)"`
35	`"library"`
36	`"parameter"`
37	`"object"`
38	`" all "`
39	`"disc"`
40	`" to "`
41	`" is "`

Related SWIs

OS_WriteC (SWI &00)

Related vectors

None

OS_PrintChar (SWI &5D)

Send a character to the printer stream

On entry	R0 = character to print
On exit	R0 = preserved
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call will send a character to the printer. OS_Bytes 3 and 5 control whether there is a printer selected and which device it is. Note that the printer ignore character (see OS_Byte 6) is not used by this call.
Related SWIs	None
Related vectors	None

OS_WriteN (SWI &46)

Write a counted string to the VDU

On entry	R0 = pointer to string to write R1 = number of bytes to write
On exit	R0, R1 preserved
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	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.
Related SWIs	None
Related vectors	WrchV

OS_SerialOp (SWI &57)

Low level serial operations

On entry

R0 = reason code
other input registers as determined by reason code

On exit

R0 preserved
other registers may return values, as determined by the reason code passed.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is like OS_Byte in that it is a single call with many operations within it. The operation required, or reason code, is passed in R0. It can have the following meanings:

- R0 = 0 – Read/write serial states
- R0 = 1 – Read/write data format
- R0 = 2 – Send break
- R0 = 3 – Send byte
- R0 = 4 – Get byte
- R0 = 5 – Read/write receive baud rate
- R0 = 6 – Read/write transmit baud rate

On the following pages is a detailed explanation of each of these reason codes in turn. Reason codes 4 and 5 can be found in the chapter entitled *Character input*.

Related SWIs

None

Related vectors

None

OS_SerialOp 0 (SWI &57)

Read/write serial status

On entry

R0 = 0 (reason code)
R1 = XOR mask
R2 = AND mask

On exit

R0 preserved
R1 = old value of state
R2 = new value of state

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The structure of this call is very similar to that of OS_Bytes between SWI &A6 and SWI &FF. The new state is determined by:

```
New state = (Old state AND R2) XOR R1
```

This call is used to read and write various states of the serial system. These states are presented as a 32-bit word. The bits in this word represent the following states:

Bit	Read/Write or ReadOnly	Value	Meaning
0	R/W	0	No software control. Must rely on hardware handshaking.
		1	Use XON/XOFF protocol. The hardware will still do CTS handshaking (ie if CTS goes low, then transmission will stop), but RTS is not forced to go low.

1	R/W	0	Use the ~DCD bit. If the ~DCD bit in the status register goes high, then cause a serial event. Also, if a character is received when ~DCD is high, then cause a serial event, and do not enter the character into the buffer.
		1	Ignore the ~DCD bit Note that some serial chips (GTE and CMD) have reception and transmission problems when this bit is high.
2	R/W	0	Use the ~DSR bit. If the ~DSR bit in the status register is high, then do not transmit characters.
		1	Ignore the state of the ~DSR bit.
3	R/W	0	DTR bit is 0.
		1	DTR bit is 1.
4 - 15			These bits are undefined. Do not modify them.
16	RO	0	XOFF not received.
		1	XOFF has been received. Transmission is stopped by this occurrence.
17	RO	0	The other end is intended to be in XON state.
		1	The other end is intended to be in XOFF state. When this bit is set, then it means that an XOFF character has been sent and it will be cleared when an XON is sent by the buffering software. Note that the fact that this bit is set does not imply that the other end has received an XOFF yet.
18	RO	0	The ~DCD bit is low, ie carrier present.
		1	The ~DCD bit is high, ie no carrier.
19	RO	0	The ~DSR bit is low, ie 'ready' state.
		1	The ~DSR bit is high, ie 'not-ready' state.
20	RO	0	The ring indicator bit in IOC is low.
		1	The ring indicator bit in IOC is high.
21 - 31			These bits are undefined. Do not modify them

Note that if XON/XOFF handshaking is used, then OS_Byte 2,1 or 2,2 must be called beforehand.

Related SWIs	OS_Byte 156 (SWI &06)
Related vectors	None

OS_SerialOp 1 (SWI &57)

Read/write data format

On entry

R0 = 1 (reason code)
R1 = –1 to read or new format value

On exit

R0 = preserved
R1 = old format value.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sets the encoding of characters when sent and received on the serial line. The bits in this word represent the following formats:

Bit	Read/Write or ReadOnly	Value	Meaning
0,1	R/W	0	8 bit word.
		1	7 bit word.
		2	6 bit word.
		3	5 bit word.
2	R/W	0	1 stop bit.
		1	2 stop bits in most cases. 1 stop bit if 8 bit word with parity. 1.5 stop bits if 5 bit word without parity.
3	R/W	0	parity disabled.
		1	parity enabled.
4,5	R/W	0	odd parity.

	1	even parity.
	2	parity always 1 on TX and ignored on RX.
	3	parity always 0 on TX and ignored on RX.
6 - 31		reserved – must be set to zero

Related SWIs

OS_Byte 156 (SWI &06)

Related vectors

None

OS_SerialOp 2 (SWI &57)

Send break

On entry	R0 = 2 (reason code) R1 = length of break in centiseconds

On exit	R0 = preserved
	R1 = preserved.

Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call sets the ACIA to transmit a break, then waits R1 centiseconds before resetting it to normal. Any character being transmitted at the time the call is made may be garbled. After sending the break the transmit process is either awakened if the buffer is not empty, or made dormant if the buffer is empty.
Related SWIs	None
Related vectors	None

OS_SerialOp 3 (SWI &57)

Send byte

On entry	R0 = 3 (reason code) R1 = character to be sent

On exit	R0 = preserved
	R1 = preserved.
	if C flag = 0 then character was sent
	if C flag = 1 then character was not sent because the buffer was full

Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call puts a character in the serial output buffer, and re-enables the transmit interrupt if it had been disabled by RISC OS. If the serial output buffer is full, the call returns immediately with the C flag set.
Related SWIs	None
Related vectors	None

OS_SerialOp 6 (SWI &57)

Read/write TX baud rate

On entry	R0 = 6 (reason code) R1 = –1 to read or 0 - 15 to set to a value
On exit	R0 = preserved R1 = old transmit baud rate
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call has the same effect as an OS_Byte 8 for writing. The value that is passed in R1 uses the same table of baud rates as this OS_Byte.
Related SWIs	OS_Byte 8 (SWI &06)
Related vectors	None

OS_WriteI (SWI &100–1FF)

Write an immediate byte to all of the active output streams

On entry	—
On exit	—
Interrupts	Interrupts are enabled Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call writes the character contained in the bottom byte of the SWI number, using OS_WriteC. It has the advantage of being more compact and quicker for a program using it than the equivalent usage of OS_WriteC. For example, to write a "J" character, you would use:

```
SWI   OS_WriteI + ASC"J"
```

Related SWIs	OS_WriteC (SWI &00)
Related vectors	WrchV

*Configure Baud

Sets the configured serial port baud rate.

Syntax

```
*Configure Baud <n>
```

Parameters

`<n>` 0 to 8

Use

This command sets the configured baud rate.

The values of n passed with this command correspond to the following baud rates:

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

The default value is 4 (1200 baud).

The receive and transmit baud rates are set from this configured value on any reset.

Example

`*Configure Baud 7` sets the configured baud rate to 9600

Related commands

None

Related SWIs

OS_Byte 7 (SWI &06), OS_Byte 8 (SWI &06),
OS_SerialOp 5 (SWI &57), OS_SerialOp 6(SWI &57)

Related vectors

None

*Configure Data

Sets the configured data protocol for the serial port.

Syntax

```
*Configure Data <n>
```

Parameters

`<n>` 0 to 8

Use

This command sets the configured protocol state in CMOS RAM.

The values of n passed with this command correspond to the following states:

n	Word length	Parity	Stop bits
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

4 (8 bits, no parity, 2 stop bits) is the default state on RISC OS.

The data protocol is set from this configured value on any reset.

Example

`*Configure Data 0` (7 bits, even parity, 2 stop bits)

Related commands

None

Related SWIs

OS_Byte 156 (SWI &06), OS_SerialOp 1 (SWI &57)

Related vectors

None

*Configure Ignore

Specifies the configured printer ignore character.

Syntax

`*Configure Ignore [<n>]`

Parameters

`<n>`	ASCII code, from 0 to 255

Use

`*Configure Ignore` specifies the configured ASCII code for the printer ignore character, used when printing is enabled via the VDU printer stream or OS_Byte 5.

The default value is 10 (ASCII linefeed). On some printers, you may find this causes lines to overprint each other, in which case you should omit the character code so all characters are sent to the printer.

The ignore character is set from this configured value on a hard reset.

Example

`*Configure Ignore 10`	Do not print ASCII character 10.
`*Configure Ignore`	Print all characters.

Related commands

None

Related SWIs

OS_Byte 246 (SWI &06), OS_Byte 182 (SWI &06), OS_Byte 6 (SWI &06)

Related vectors

None

*Configure Print

Selects the default destination for printed output.

Syntax

```
*Configure Print <n>
```

Parameters

`<n>`	0 to 7

Use

This command sets the default printer destination in CMOS RAM.

The values of `<n>` passed with this command correspond to the following printers:

n	**Printer**
0	Printer sink (no output)
1	Parallel port
2	Serial port
3	User printer driver
4	Network printer (handled through NetPrint)
5-7	Files in system variables PrinterType$<5|6|7>

This option can also be set to 0, 1, 2 or 4 from the desktop, using the Configure application.

The default destination is set from this configured value on a hard reset.

Example

```
*Configure Print 1        select the parallel printer port
```

Related commands

None

Related SWIs

OS_Byte 5 (SWI &06)

Related vectors

None

*Ignore

Sets the printer ignore character to the given ASCII code.

Syntax

`*Ignore [<number>]`

Parameters

`<number>`	an ASCII code, a number from 0 to 255

Use

`*Ignore` specifies the ASCII code for the printer ignore character, used when printing is enabled via the VDU printer stream or OS_Byte 5.

The default value is 10 (ASCII linefeed). On some printers, you may find this causes lines to overprint each other, in which case you should omit the character code so all characters are sent to the printer. `*Ignore 0` will not ignore all characters; it will ignore the null character.

OS_Byte 246 will read and write this value. OS_Byte 182 controls the NoIgnore state. OS_Byte 6 performs the same action as this command.

Example

`*Ignore 10`	Do not print ASCII character 10.
`*Ignore`	Print all characters.

Related commands

None

Related SWIs

OS_Byte 5 (SWI &06), OS_Byte 246 (SWI &06), OS_Byte 182 (SWI &06), OS_Byte 6 (SWI &06)

Related vectors

None

*Spool

Sends everything appearing on the screen to the named file

Syntax

`*Spool [<pathname>]`

Parameters

`<pathname>`	a valid pathname specifying a file

Use

`*Spool <pathname>` opens the specified file for output. Any existing file of the same name will be overwritten. All subsequent characters sent to the VDU drivers will also be copied to the file, using OS_BPut. (If OS_BPut returns an error, the spool file is closed (thereby restoring the spool handle location) and this error is returned from OS_WriteC.)

This continues until either a `*Spool` command (with or without a file name) is issued or a `*SpoolOn` is issued which will terminate spooling started by `*Spool`.

If the pathname is omitted, the current spool file, if any, is closed, and characters are no longer sent to it. If the pathname is given,. then the existing spool file is closed and the new one opened.

You can temporarily disable the spool file, without closing it, using OS_Byte 3.

Example

```
*Spool %.Showdump
*Spool
```

Related commands

*SpoolOn

Related SWIs

OS_Byte 3 (SWI &06), OS_Byte 199 (SWI &06) OS_BPut (SWI &0B) OS_File (SWI &08)

Related vectors

ByteV, BPutV

*SpoolOn

Add data sent to the screen to the end of an existing file

Syntax

`*SpoolOn [<pathname>]`

Parameters

`<pathname>`	a valid pathname specifying a file that exists

Use

`*SpoolOn <pathname>` is similar to *Spool, except that it takes the name of an existing file. An error will occur if the name of a non-existent file is given. All subsequent characters sent to the VDU drivers will also be copied to the end of the file, using OS_BPut. (If OS_BPut returns an error, the spool file is closed (thereby restoring the spool handle location) and this error is returned from OS_WriteC.)

This continues until either a `*SpoolOn` command (with or without a file name) is issued or a `*Spool` is issued which will terminate spooling started by `*SpoolOn`.

If the pathname is omitted, the current spool file, if any, is closed, and characters are no longer sent to it. If the pathname is given, then the existing spool file is closed and the new one opened for appending.

You can temporarily disable the spool file, without closing it, using OS_Byte 3.

Example

```
*SpoolOn %.Showlist
*SpoolOn
```

Related commands

*Spool

Related SWIs

OS_Byte 3 (SWI &06), OS_Byte 199 (SWI &06) OS_BPut (SWI &0B) OS_File (SWI &08)

Related vectors

ByteV, BPutV

VDU Drivers

Introduction

Though strictly speaking part of the character output system, 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
- graphics and text windows
- colour palette
- colour patterns
- the mouse
- putting text and graphics on the screen
- multiple display pages

The chapter entitled *character output* described how to write to the VDU. This chapter describes what special effects occur when particular characters are sent.

There are also a large number of VDU specific commands that allow fine control of its operation.

There are five important aspects of VDU interaction which are not described in this chapter. These are:

- the Font manager
- the Window manager
- the Draw module
- Sprites
- the ColourTrans module

These are implemented as modules separate from the RISC OS kernel, and are described in their own chapters.

Overview

The most important call relating to the VDU is OS_WriteC, as this is used in nearly all programs which have to output to the screen. Other calls can be used for more direct control of the VDU facilities.

The VDU display on RISC OS comes from the VIDC chip. This reads the contents of a block of memory and converts it into a form that can drive a video monitor.

VDU commands

This chapter differs from others in this manual in that, in addition to a list of SWIs and *commands, there is also a list of VDU commands. To issue VDU commands, simply use OS_WriteC to send characters to the VDU stream. All characters are strictly VDU commands, but those between 0 and 31, and 127 are of special interest because they cause special actions to take place. The others are simply printed on the screen as a character.

These special characters are used as commands. They can be followed by a sequence of characters, the length of which depends on the command. In some cases, the character on its own is sufficient, but it can require up to 9 following bytes to complete the command. These bytes are queued until the required number are in the queue before the command is executed.

To represent these sequences of characters sent to the VDU using OS_WriteC, a shorthand is used in this chapter. You will see VDU followed by numbers separated by commas. This represents each character being sent through OS_WriteC.

For example, VDU 65 sends character 65, an ASCII 'A', to OS_WriteC. VDU 17,3 sends character 17 followed by character 3.

Modes

RISC OS supports many different ways of displaying information on the screen. Each of these different ways is called a mode. The exact number of modes available depends 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. Two main characteristics distinguish the modes.

- The resolution of a mode relates to the number of pixels which can be displayed horizontally and vertically.

- The number of colours that can be displayed at once is determined by the number of bits used to store each pixel. Typically, 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.

A complete list of the available modes is given in the description of VDU 22, which is the command that changes modes.

Text and graphics

There are two distinct types of object that the VDU drivers can draw onto the screen.

- The text VDU deals with drawing text characters
- The graphics VDU handles any arbitrary drawing of dots, lines, shapes, etc.

Windows

Different commands will act to either text or graphics areas. Each has a window, or area where their output will go. After a mode change, both text and graphic windows fill the screen and overlap each other exactly. There is no conflict in having them overlap, since the window is just a declaration of boundaries. Either window can be changed at any time to be any size. Any output to a window will be clipped to it. For example, if only part of a line appears in the graphics window, then only that part will be shown and the rest ignored.

A cursor is the place at which the next output will go. There are independent text and graphics cursors, which must remain inside their relevant window.

Various control commands are provided to affect the output in text and graphics windows. Examples of such actions are:

- changing the colours in which output occurs,
- moving the appropriate cursor,
- clearing the window.

Text VDU

Text characters are patterns of pixels which are positioned on the screen at character-aligned positions. That is, the screen is treated like an array of character sized boxes, into which can go any printable character.

All text display is normally confined to the text window. All scrolling is confined to this region, sometimes called the scrolling window, because text can be scrolled within it. The graphics window cannot be scrolled automatically; but you can use block move to perform scrolling.

The text cursor shows the position on the screen of the next character to be displayed. This is usually a flashing underline. There can be a second cursor which is used with cursor editing (this is described later).

Note that there are some screen modes that will only display text.

Graphics VDU

The graphics VDU handles the drawing of objects such as points, lines, circles, ellipses, etc. The graphics window, like the text window, starts as the whole screen after a mode change. The graphics cursor, which is invisible, marks the last point at which a graphics operation ended.

Joining text and graphics

The VDU driver can be configured to print text at the graphics cursor instead of the text cursor. This means that text will be drawn using the current graphics cursor for positioning, and using the graphics colour, etc. 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 disadvantages are that the characters take longer to draw and scrolling is not available. Generally, when text is printed at the graphics cursor, this is referred to as VDU 5 mode because this is the command that enables it.

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, but 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 usual rate. The Copy key has the action of copying what is under the input cursor to the output cursor as if it was typed.

See the chapter entitled *Character input* for a full description of these keys and their control.

Cursor editing mode 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 Return at the end of an input line.

Colours

The number of colours available on the screen at any time is either 2, 4, 16 or 256. When you first enter a mode, the default colours are assigned. These can subsequently be changed with the palette.

256-colour modes

In 256 colour modes, there are 64 different colours, and each colour may have four different shades, resulting in a total of 256 different colours.

Foreground and background

You may choose to display your text or graphics in a different colour from the defaults. To do this, there are commands to change the foreground and background of each. Usually, the foreground colour is that in which the text or graphics drawing is done, and the background colour is used for all other drawing, such as a screen clear. RISC OS can be changed so that the background colour is used for drawing if required.

The palette

Another important part of the VDU is the palette. This is the control of what colours appear on the screen. The palette is a table built into the VIDC chip which determines the relationship between the colour number stored in the screen memory (logical colour), and the actual colour information sent to the monitor (physical colour). Care should be taken not to confuse logical and physical colours. Thus, while colour 0 on RISC OS is black by default, it can be made to be any colour by changing how the palette maps it.

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 (2^{12}) physical colours. Flashing colours are accomplished by a logical colour having 2 physical colours associated with it. These are swapped at a programmable rate, causing flashing.

The palette also controls the colour of the border around the screen and the colours of the mouse pointer. These can be set independently of any other colour on the screen. The border and mouse colours are always 2bpp (4 colours) in all screen modes.

Tints

In 256 colour modes, each pixel is represented by an 8-bit value. Six bits are the logical colour, and the other two bits are the tint. The tint is is a direct control of the amount of grey which is added to the base colour, to one of 4 levels.

The six bits in the logical colour set the basic colour from the range of different shades of colours provided by the palette. The tint is the fine control within this range.

ECF patterns

The Extended Colour Fill patterns are a means of increasing the apparent number of colours by producing a fine chequerboard mix of colours. This is of most use in modes where there are few colours available, because it gives the effect of having more colours on the screen than there are.

Four different ECF patterns are provided, and can be independently defined.

Normally, the origin of the ECF patterns is based on the bottom left corner of the screen. This can be changed, so that it aligns with any point on the screen, such as the current graphics window.

Bell

The VDU drivers control how the bell will sound. The bell is a sound that is made when the standard ASCII character 7 (Ctrl-G) is sent to the VDU. Its volume, pitch and duration can all be customised.

Mouse and pointer

The mouse is a device that is moved on a surface, rolling an internal ball, usually with several buttons. The pointer is a reflection on the screen of the mouse's movements. Normally, it appears as a small arrow, but can be programmed to be any shape. It is also possible to disconnect the pointer from the mouse and move the pointer to where the program wants it to be. This is useful when switching between windows under program control.

RISC OS provides control over how much the pointer moves in response to a mouse movement. This sensitivity control can be useful in situations where fine or coarse movement is required by different programs.

Screen configuration

Full control is given over how video information is generated. Depending on how it looks on screen, the display can be shifted up or down. Some monitors do not allow for this adjustment, so this facility is provided.

Also, the interlace can be switched on or off. Interlace means that images sent to a monitor alternate one scan line up and down on alternate frames. On a monitor which has a long persistence phosphor (images take some time to fade), an interlaced image eliminates the 'lined' effect of a screen image. On a short persistence screen interlace can cause a flicker, because the first image has faded before the second one is finished.

RISC OS supports many different kinds of monitor. Depending on the type of monitor used, only a subset of all possible modes are available on it. Thus there is a command to set which monitor is connected, so that incorrect modes are not accidentally entered.

Multiple banks

Normally, there is one bank of memory that is used for the screen. If it is changed, then this is reflected on the screen as it is refreshed by VIDC. Sometimes it is useful to write to one bank of screen memory, while another is displayed and then swap when finished. This produces an 'instant draw' effect, which is visually pleasing.

Whilst normally only two banks would be used for this kind of application, you can have as many banks as will fit in the allocated screen RAM area. This requires copies of the screen RAM requirements for each bank. For example, with two banks of screen memory, an 80K mode will require 160K.

Writing to the screen

Many different kinds of things can generate output on the screen, or more strictly speaking, the current screen bank. Text or graphics can be written, and many commands exist to alter where and how output will appear on the screen.

Writing text

Sending printable characters through OS_WriteC will result in it appearing at the text cursor position in the current window. It will wrap around to following lines when it reaches the right hand side of the window. Certain control commands can move the text cursor in all directions or to a given place in the window. Usually, the cursor moves right after a character is printed. This can be changed so it moves in any of the four directions.

Writing graphics

Many different kinds of graphics can be put onto the screen, such as:

- circles, ellipses, arcs, segments, and sectors
- triangles, rectangles and parallelograms
- filled areas, such as all those above and any irregular shape
- dots
- solid and dotted lines
- text in VDU 5 mode

See the chapter on sprites to see how any sized array of pixels can be written to the screen.

As well as different shapes, there is control over how it is written over what is already on the screen. It can be configured to:

- overwrite existing graphics,
- OR with it,
- AND with it,
- exclusive OR with it,
- invert it,

and so on.

As well as having control over the colour and writing mode, you can use any of the ECF patterns to write with.

Clearing the screen

The graphics or text windows can either be completely or partially cleared. This will be done with the current graphics or text background colour as appropriate.

Synchronised writing

There is a mechanism under RISC OS of waiting until a Vsync event occurs and then writing to the screen. This can make screen update very smooth, as writing to the screen memory does not clash with the VIDC chip reading it to send to the monitor. If they do clash, then a 'tearing' can appear briefly. This is because one part of the memory being written to is displayed in its old state and the other part in the new.

Unless you plan to use multiple paging techniques, then this is a good way of achieving smooth animation.

Reading from the screen

As well as writing to the screen, it is possible to read some information back from it. There is a command to read a character from under the text cursor and work out what its ASCII value is. Cursor editing uses this facility.

You can also read the logical colour and tint of a point. Given that there is another call to return the palette setting for a colour, it is easy to combine the two and work out the 'real' colour of pixels on the screen.

The screen can be saved as a file, which can be subsequently treated as a sprite, or edited with Paint for example. There is a corollary command to load it back onto the screen.

Information about the VDU

There are a number of calls to get all kinds of information about the configuration and status of the VDU driver. Here is some of the information that can be read:

- size and position of graphics and text windows
- position of graphics and text cursors
- description of current screen mode
- size of screen memory
- palette mapping
- foreground and background text and graphics colours
- banks used by VDU and screen
- number of bytes queued for a VDU command being composed
- number of lines printed since last page halt
- in VDU 5 mode or not

VDU extension vector

The normal VDU driver can be completely replaced with a custom driver if required. The VDU extension vector, called VDUXV, can be called instead of the normal VDU vector. This can be useful if you want to change the characteristics of screen output in a dramatic way. For example, the font manager module uses this to quickly display complex fonts. Going through the normal VDU mechanism would be too slow, because it would have to be done a dot at a time.

Technical Details

VDU commands

As mentioned earlier, 'VDU' followed by a series of numbers separated by commas is used in this chapter to represent a character being sent to OS_WriteC. For convenience, we will use the shortcuts that BBC BASIC uses with its VDU statement. Here is a brief reminder of the syntax of that statement:

VDU n sends character 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. coordinates 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.

Screen modes

When changing mode, a great many things are initialised. For a complete list of these and other mode notes, see VDU 22.

*Configure Mode will set up the screen mode to be used after a hard reset.

When a program wishes to change mode, it must check that there is enough memory allocated for the screen for that mode and that the monitor being used is compatible with the mode. OS_CheckModeValid must be called to check these two things. If you don't, then VDU 22 will do it anyway, but it is better for the program to be aware of what's happening. If the mode requested cannot be used, OS_CheckModeValid will also return a suggestion for a mode to use in place of it.

Screen configuration

The computer can adjust its output to suit its attached monitor in a number of ways.

*Configure MonitorType is used to tell RISC OS what kind of monitor is attached, since the system has no way of detecting this from hardware. This command allows the system to subsequently disallow any modes that are not compatible with the attached monitor.

*Configure Sync will set up the vertical sync output of the video connector to be vertical or composite sync. Different monitors may require either of these, though most use composite sync.

*Configure TV, *TV and OS_Byte 144 can all adjust the position of the video output up or down by several lines, and switch interlace on and off. VDU 23,0 can also control the interlace setting.

Multiple bank modes

There are two main commands that can be used to handle multiple banks of screen memory. OS_Byte 112 selects which bank of memory to send VDU output to. OS_Byte 113 selects which bank of memory is used by the VIDC hardware to write out to the screen. By using these two, it is simple to swap screens at will.

OS_Byte 250 reads the current OS_Byte 112 setting, and OS_Byte 251 reads the current OS_Byte 113 setting.

In order to use multiple banks, you will probably have to use *Configure ScreenSize to set the amount of memory to reserve for all the banks.

*Shadow exists mainly for compatibility with BBC/Master operating systems. Under RISC OS, it can select between two banks of memory to be used on the next mode change. OS_Byte 114 has the same effect as *Shadow. OS_Bytes 112 and 113 support the shadow system, but you are better off using bank numbers directly.

For those who want low level access to screen banks, OS_Word 22 allows setting the addresses of the VDU bank and the VIDC bank directly.

Colours

These are the colours as set up after a mode change:

Two-colour modes

0 = black
1 = white

Four-colour modes

0 = black
1 = red
2 = yellow
3 = white

16-colour modes

0 = black
1 = red
2 = green
3 = yellow
4 = blue
5 = magenta
6 = cyan
7 = white
8 = flashing black-white
9 = flashing red-cyan
10 = flashing green-magenta
11 = flashing yellow-blue
12 = flashing blue-yellow
13 = flashing magenta-green
14 = flashing cyan-red
15 = flashing white-black

256-colour modes

256 colour modes are treated differently from the others. Instead of using the standard 16 entry physical colour table, there are two systems which are used by different commands. The internal format is the less easy to use of the two. In it, the bits are structured as follows:

Bit	Meaning
0	Bit 0 of palette index
1	Bit 1 of palette index
2	Bit 2 of palette index
3	Bit 3 of palette index
4	Red bit 3 (high)
5	Green bit 2
6	Green bit 3 (high)
7	Blue bit 3 (high)

where the palette index (0 - 15) controls which VIDC palette entry is used, but with some bits of the palette entry then being overridden by the top 4 bits of the memory byte. With the default palette setting, this becomes:

Bit	Meaning
0	Tint bit 0 (red+green+blue bit 0)
1	Tint bit 1 (red+green+blue bit 1)
2	Red bit 2
3	Blue bit 2
4	Red bit 3 (high)
5	Green bit 2
6	Green bit 3 (high)
7	Blue bit 3 (high)

Each primary colour has 4 bits of intensity, but the two least significant bits (the tint bits) are shared between the three colours. Therefore, some intensities of a primary colour (for example, red) can only be obtained at the expense of adding in a certain amount of grey.

The second form for 256 colours, which is used by some commands is structured as follows:

Bit	Meaning
0	Red bit 2
1	Red bit 3 (high)
2	Green bit 2
3	Green bit 3 (high)
4	Blue bit 2
5	Blue bit 3 (high)
6	Tint bit 0 (red+green+blue bit 0)
7	Tint bit 1 (red+green+blue bit 1)

The tint is controlled separately in most commands; bits 6 and 7 are only used in the native ECF setting, which is not often used in 256 colour modes.

This format is converted into the internal format when stored, because that is what the VIDC hardware recognises.

To change colour

VDU 17 can be used to change the text colour. VDU 23,17,5 I will exchange the text foreground and background colours.

VDU 18 can change the graphics colour, and much more than just that. Because graphics can interact with what is already is on the screen, then VDU 18 can set up the graphics to be ORd, ANDed, XORd, inverted and so on.

In 256 colour modes, VDU 23,17,0 - 3 can be used to set the tints to be used when next printing/plotting.

Palette

VDU 19 can be used to change the way that the palette defines the logical to physical colour relationship. It has many modes and as well as changing the logical colours, can also set the border, flashing and cursor colours. OS_Word 12 can also be used to write the palette.

VDU 20 will return the palette to the condition that it was just after a mode change. This would be used by a program just before finishing, if it had altered the palette during running.

If you want to read the palette setting of a colour, OS_ReadPalette or the BBC/Master compatible OS_Word 11 can be used.

Flashing colour

RISC OS will swap two colours at a programmed interval. If they are the same colour, then there is no noticeable effect. If they are different, then flashing will result. VDU 19 can individually set these colours to be any colour from the palette.

The speed at which flashing occurs can be controlled by OS_Bytes 9 and 10. They set the duration in video frames. VDU 23,9 and VDU 23,10 have the same effect as these calls. The duration settings can be read by OS_Bytes 194 and 195.

OS_Byte 193 allows a program to read or alter the flash counter. This is a decrementing counter that swaps colours when the count reaches zero.

ECF patterns

There are several different ways of changing ECF patterns. The main command is VDU 23,2-5. This can operate in two modes depending on the setting of VDU 23,17,4. Also, VDU 23,12-15 can be used for simpler patterns.

Colours and resolution

Both commands are passed 8 bytes that define the pattern. The number of pixels depends how many colours are available in the screen mode you are using:

Colours available	**Number of pixels set by each line** VDU 23,2-5	**VDU 23,12-15**
2	8	2
4	4	2
16	2	2
256	1	1

You can see that while the number of pixels in the pattern diminishes, the number of potential colours increases.

256 colour patterns

As you can see, in a 256 colour mode, the pattern is simply a colour description for each line. VDU 23,2-5 uses the internal 256 colour map, while VDU 23,12-15 uses the simpler colour map. When stored, the internal form is used. This should be borne in mind if you use OS_Word 10 to read the ECF definitions.

VDU 23,12-15

This call uses a simpler pattern. The 8 parameters passed form a pattern as follows:

1	2
3	4
5	6
7	8

So it describes a simple 2 by 4 pattern for all but 256 colour modes. Here it is one colour per line, for all 8 lines, like VDU 23,2-5.

VDU 23,2-5

This call is more complex. It uses one line per parameter, and there is a direct tradeoff between colours and resolution. Thus, for a 2 colour mode, it can display an 8 by 8 pattern of on or off pixels, in 256 colour mode, it can only generate 8 lines of a single different colour each.

VDU 23,17,4

VDU 23,17,4 is used to select between BBC/Master compatible mode and native RISC OS mode. These modes describe how ECF colour descriptions are mixed when using VDU 23,2-5. For some examples, see the *Application Notes* at the end of this chapter.

Initialisation

VDU 23,11 will reset the ECF pattern definitions to their default values. It will also reset the VDU 23,17,4 flag to the default BBC/Master compatible state.

Setting the origin

By default, patterns are written as if their bottom left hand corner aligned with the bottom left hand corner of the screen. Using OS_SetECFOrigin, you can adjust this to be any point on the screen. Thus, an ECF pattern can now be aligned with any object, such as the graphics window. VDU 23,17,6 has the same effect as this call.

Bell

The bell can be made to sound by sending a VDU 7 to OS_WriteC.

To configure how it will sound:

- OS_Byte 211 will select the sound channel used
- OS_Byte 212 will adjust the volume
- OS_Byte 213 will adjust the frequency
- OS_Byte 214 will adjust the duration

*Configure Quiet will select a medium volume, while *Configure Loud will select the loudest volume.

Cursors

VDU 5 will link text and graphics cursors and cause all subsequent output to be printed at the graphics cursor position. This command can be cancelled using VDU 4. The text input cursor is normally displayed unless disabled by VDU 23,1. Both this and VDU 23,0 can be used to change the appearance of the cursor.

There are a number of VDU commands that affect the position of the text cursor directly:

- VDU 30 – send the text cursor to its home position, which is usually the top left corner of the current window.
- VDU 31 – set the text cursor to any position on the screen.
- VDU 8 – back space
- VDU 9 – horizontal tab
- VDU 10 – line feed. That is, move down.
- VDU 11 – vertical tab. That is, move up one line.
- VDU 13 – move back to the start of the line.
- VDU 127 – delete. That is, backspace, print a space then backspace again

The position of the text cursor can be read with OS_Byte 134. If cursor editing is in progress, then OS_Byte 165 can be used to read the position of the output cursor, usually displayed as a solid blob.

Normally, when a character is printed, the cursor currently used will move to the right. This action can be controlled by VDU 23,16. It can set the cursor to move in any of four directions. It also controls how cursors act at the end of lines, and so on.

OS_RemoveCursors will remove the input and output cursors and store their state internally. A subsequent call to OS_RestoreCursors will restore them exactly. These calls are used mainly by low-level draw routines to avoid mixing the cursors with what is drawn on the screen.

OS_Word 13 will return the current and previous graphics cursor positions. Using OS_ReadVduVariables, even earlier coordinates can be read.

Mouse and pointer

When a mouse button is pressed or released a record is kept in the mouse buffer. OS_Mouse will read a mouse record from this buffer. It stores the position of the mouse, the state of its buttons and the time the record was put into the buffer. OS_Byte 128 can also be used for this as well as reading how much free space is in the mouse buffer.

OS_Word 21,3 will set the mouse position, so subsequent writes to the mouse buffer will assume the mouse is at the specified location, and move from there.

OS_Word 21,4 will read the unbuffered mouse position. That is, where it is at the moment of calling this function. This bypasses the buffer, so subsequent reads of the buffer may not tie up with this position. It is better to use one or the other method exclusively in a program.

Pointer

The ratio of mouse movement to pointer movement on screen can be controlled by OS_Word 21,2 or permanently set by *Configure MouseStep.

The pointer that appears on the screen can be defined in four shapes. OS_Word 21,0 can define the shape and colour of each of these. OS_Byte 106 is used to select which pointer to use, or switch it off completely. *Pointer can also be used to switch it on or off.

The pointer will be confined to the box defined by OS_Word 21,1. This would usually be set to the graphics window.

The pointer's position on the screen can be set with OS_Word 21,5 and read with OS_Word 21,6.

Getting information

There are many ways of extracting information about the state and configuration of the VDU system.

OS_Byte 217 will read the number of lines since the display was last stopped scrolling if it was in paged mode.

OS_Byte 218 returns how many bytes are in the VDU queue. This is used when a multiple byte VDU command is being collected.

OS_Byte 163 will return the current dot-dash line length and the amount of memory allocated for sprites. It can also set the dot-dash length.

OS_ReadDynamicArea is a better way to read the amount of memory allocated for system sprites – this call will also return the memory allocated for screen bank use.

OS_Byte 117 reads the VDU status. This involves:

- whether the printer output is enabled
- if paged scrolling is enabled
- if in shadow mode

- if in VDU 5 mode
- if cursor editing
- if the screen is disabled with VDU 21

OS_ReadVduVariables provides a large number of variables that can be read. OS_Byte 160 is a subset of this, kept for BBC/Master compatibility reasons. Almost all information about windows, cursors and colours can be accessed here. Two special variables provided are a pointer to a fast horizontal line draw routine and access to colour blocks.

OS_ReadModeVariable returns the fixed information about a mode, such as how many pixels across and down it is, and how many colours it supports.

Reading from the screen

OS_Byte 135 will read the ASCII value of the character at the text cursor position and also reads the current screen mode.

OS_ReadPoint will read the logical colour of a pixel. OS_Word 9 performs much the same function, but is kept mainly for compatibility with BBC/Master series.

*ScreenSave will copy the screen contents into a file where it can subsequently be edited with Paint or reloaded to the screen with *ScreenLoad.

Writing to the screen

Output to the screen can be disabled by VDU 21. It can be restored by VDU 6.

VDU 26 will restore the graphics and text windows to their default states. That is, both filling the screen.

Text

Text can be sent to the screen with any VDU command from 32 to 255, excepting 127 which is the delete command.

VDU 28 defines the text window. VDU 12 will clear the window that the text cursor is in. After a VDU 12, the text cursor is moved to its home position, usually the top left hand corner. VDU 23,8 will clear a block within the text window.

Page mode means that when about 75% of a screenful has been shown, then the system will pause and wait for Shift to be pressed before starting again. This stops text being lost from scrolling off the top of the screen too quickly Paged mode can be enabled by VDU 14 and disabled with VDU 15. By default, paged mode is off.

*Configure Scroll and NoScroll configure whether text will scroll when it reaches the bottom of the text window. This means that when NoScroll is set a character can be printed at the bottom right of the screen without immediately scrolling the screen. This feature can also be controlled with VDU 23,16 and allows a full screen of text to be simply printed.

VDU 23,7 can scroll the text window or the whole screen in any direction.

In VDU 5 mode, it is possible to change the size and spacing of text with VDU 23,17,7. This is how you would generate a message with large gaps between the characters.

Redefining characters

Each printable character (one that is not a command) is an array of 8 by 8 pixels that is defined in the shape of standard ASCII and ISO characters. All of these characters can be redefined to be any pattern.

To change the definition of a printable character, VDU 23,32-255 must be used. The character number that you wish to redefine is the second parameter, in the range 32-255. It is followed by 8 bytes that define the bit pattern to be used.

OS_Byte 20 will reset all character definitions to their default. OS_Byte 25 will reset a given group of them. OS_Word 10 can read the definition of any character from the current system font.

Printer

VDU 1 will send the following character to the printer stream. VDU 2 will enable the stream, so that all characters sent to the VDU are also sent to the printer stream. This state can be disabled by VDU 3.

Graphics

VDU 24 will define the position of the graphics window. VDU 16 will clear it to the current graphics background colour.

VDU 25 is the main graphics plot command. OS_Plot has the same effect as it, but is much faster, avoiding the delays inherent in the VDU stream. They both have a type parameter followed by x and y coordinates. The type covers moving the graphics cursor, plotting points, lines (solid and dotted), triangles, rectangles, parallelograms, circles, arcs, sectors, segments, ellipses and other graphic forms. These figures can be hollow or filled with the graphics foreground colour. It handles relative or absolute drawing That is, the x and y are relative to the current x and y or moving to a new absolute position on the screen.

When plotting dotted lines, the default pattern is a dot-space pattern repeated. This can be changed to any pattern. VDU 23,6 is passed 8 bytes that define a pattern up to 64 bits in length to be repeated. OS_Byte 163 sets how many bits are to be used. Simple patterns like &FF (solid line), &AA (the default dot-space) and &EE (dashed line : dot-dot-dot-space) can be used or any more complex pattern up to 64 bits in length. OS_Word 10 can read the current definition.

VDU 29 sets the graphics origin. This is the point on the screen that becomes the 0,0 point for all subsequent graphics operations.

OS_ChangedBox will tell you what area of the screen has been changed. This can be used to reduce the amount of redrawing that needs to be done by an application.

*ScreenLoad complements *ScreenSave, discussed earlier and load a file into the screen memory.

Vsync

OS_Byte 19 will wait until a Vsync occurs before returning. This allows programs that are quick enough to write to the screen without any kind of flickering or tearing of images.

VDU 0

Null Operation

Syntax

VDU 0

Parameters

—

Use

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

Syntax	VDU 1,<character>
Parameters	<character> to send to the printer stream
Use	VDU 1 sends the next character to the printer stream 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.
Example	`VDU 1,10` Send a line feed to the printer stream, if enabled

VDU 2

Enable printer stream

Syntax

VDU 2

Parameters

—

Use

VDU 2 enables the printer stream. After this call, most characters sent to the screen will also be sent to the currently selected printer device. OS_Byte 5 controls this, and is described in the character output chapter. 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 the argument of 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 3, 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 117.

VDU 3

Disable printer stream

Syntax

VDU 3

Parameters

—

Use

VDU 3 cancels the effects of VDU 2 so that all subsequent printable characters are not passed through the kernel printer driver.

VDU 4

Split cursors

Syntax

VDU 4

Parameters

—

Use

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 23,1) 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 23,16.

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), providing scrolling is enabled. Cursor editing is allowed in this mode.

You can determine whether the cursors are split or joined using OS_Byte 117.

VDU 5

Join cursors

Syntax

VDU 5

Parameters

—

Use

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 topmost row, lefthand edge of the character being placed there. Characters are displayed in the current graphics foreground colour using the current graphics action. The background pixels in the character shape are not plotted.

You can set the character sizing and spacing using VDU 23,17,7…

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 23,16). 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

Syntax

VDU 6

Parameters

—

Use

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 117.

VDU 7

Bell

Syntax

VDU 7

Parameters

—

Use

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 211 - 214.

VDU 8

Back space

Syntax

VDU 8

Parameters

—

Use

VDU 8 causes either the text cursor (by default) 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 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

Syntax

VDU 9

Parameters

—

Use

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

Syntax

VDU 10

Parameters

—

Use

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

Syntax

VDU 11

Parameters

—

Use

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

Syntax

VDU 12

Parameters

—

Use

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

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

VDU 13

Carriage return

Syntax

VDU 13

Parameters

—

Use

VDU 13 causes the text cursor or, in VDU 5 mode, the graphics cursor 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 character 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

Syntax

VDU 14

Parameters

—

Use

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 117 may be used to determine whether page mode is enabled. See also OS_Byte 217.

VDU 15

Page mode off

Syntax

VDU 15

Parameters

—

Use

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

VDU 16

Clear graphics window

Syntax

VDU 16

Parameters

—

Use

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

Set text colour

Syntax

VDU 17,<colour>

Parameters

<colour>	logical text colour

Use

VDU 17 is used to assign a logical colour to either the text foreground or background according to the value of colour, 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 – 64 in 256 colour mode) 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 colour depends on the type of mode:

Colours	colour parameter meaning	
2,4,16	Logical colour for that pixel	
256	Bottom 6 bits of colour 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.

Example

`VDU 17,12` Set to logical colour 12

VDU 18

Set graphics colour and action

Syntax

VDU 18,<action>,<colour>

Parameters

<action>	operation to perform
<colour>	colour to use

Use

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 action as follows:

Value	**Action**
0	Overwrite colour on screen with colour
1	OR colour on screen with colour
2	AND colour on screen with colour
3	exclusive OR colour on screen with colour
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND colour on screen with (NOT colour)
7	OR colour on screen with (NOT colour)
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 - 95	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 transparency 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 on 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 colour 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 (as for VDU 17).

Where action has specified a colour pattern, then colour 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 colour 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.

Example

`VDU 18,1,6`	Write, ORing with the screen in colour 6

VDU 19

Set palette

Syntax

VDU 19,<logical colour>,<mode>,<red>,<green>,<blue>

Parameters

<logical colour>	colour to set
<mode>	how to set the colour
<red>,<green>,<blue>	physical colour information

Use

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

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

mode = 0 - 15	logical colour = actual colour red, green and blue are ignored
mode = 16	logical colour = red units red green units green blue units blue This sets both flash palettes for logical colour
mode = 17	Defines first flash palette for logical colour
mode = 18	Defines second flash palette for logical colour
mode = 24	Defines border colour = red units red green units green blue units blue logical colour is not used
mode = 25	Define logical colour (1 - 3) of pointer = red units red green units green blue units blue

If you add 128 to the 'mode' value, you also set the 'supremacy' bit of the appropriate palette entry. This is used when the computers' 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 bottom nibble may be significant in future versions of the hardware – to cater for this you should replicate the top nibble in the bottom nibble, by multiplying each RGB component by 17/16. Therefore, &F0F0F000 becomes &FFFFFF00.

In normal non-flashing colours, what this means is that both of the flash colours are the same. RISC OS will swap colours at a programmed interval. If they are the same colour, then there is no noticeable effect. 'mode' values of 17 and 18 allow any colour to be made to flash with any combination of colours.

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 the 'mode' value is in the range 0 - 15. The actual colour number depends on the logical colour and the number of bits per pixel used in a given screen mode as follows:

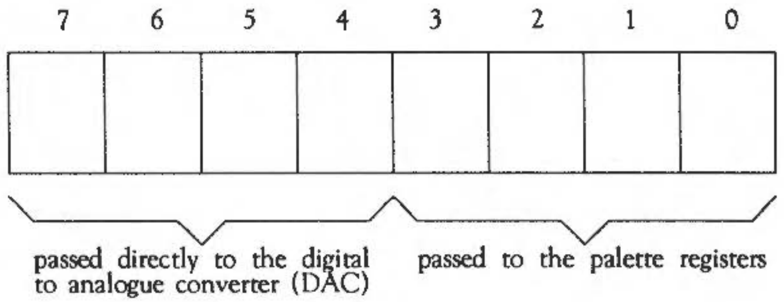
Logical	**Bits per pixel in a screen mode**		
colour	**1**	**2**	**4**
0	0	0	0
1	7	1	1
2		3	2
3		7	3
4			4
5			5
6			6
7			7
8			8
9			9
10			10
11			11

12	12
13	13
14	14
15	15

The meanings of the mode 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:


7
6
5
4
3
2
1
0
passed directly to the digital to analogue converter (DAC)
passed to the palette registers

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 12, and read the current palette using OS_Word 11 and OS_ReadPalette.

Example

`VDU 19,5,12,0,0,0`	Set logical colour 5 to be physical colour 12

VDU 20

Restore default colours

Syntax

VDU 20

Parameters

—

Use

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

Syntax

VDU 21

Parameters

—

Use

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 character 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 117 to determine whether the VDU driver is currently enabled or disabled.

VDU 22

Change display mode

Syntax

VDU 22,<mode>

Parameters

<mode>	the screen mode to select

Use

VDU 22 is used to select a screen mode. The modes available depend on the configured monitor type (see *Configure MonitorType in this chapter). The three types are:

Type	Monitor
0	'TV' type monitor
1	multi-sync monitor
2	61.2 kHz high-resolution monochrome monitor
3	60Hz VGA-type monitor

The bottom seven bits of mode are used to select the mode.

Below is a list of the modes available to type 0 and 1 monitors.

- The refresh rate is 50Hz, or 60Hz for modes 25 - 28
- Modes 3, 6 and 7 do not display graphics for compatibility with BBC/Master series computers.
- Modes 11, 14 and 17 are not a multiple of eight pixels high. By default, in these modes the bottom of the screen corresponds to the bottom line of ECF patterns, but the top line will not correspond to the top line of ECF patterns.
- In modes 16, 17 and 24 circles, arcs, sectors and segments do not look circular. This is because the aspect ratio of the pixels is not in a 1:2, 1:1 or 2:1 ratio.

Mode	Text col x row	Resolution hor x ver	Logical colours	Bits per 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	640 x 250	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	640 x 250	16	4	80K
15	80 x 32	640 x 256	256	8	160K
16	132 x 32	1056 x 256	16	4	132K
17	132 x 25	1056 x 250	16	4	132K
24	132 x 32	1056 x 256	256	8	264K

There are further modes which are only available for use on monitor type 1 (multi-sync). Modes 25 - 28 are the VGA modes and can also be used with a type 3 monitor.

Mode	Text col x row	Resolution hor x ver	Logical colours	Bits per 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
25	80 x 50	640 x 480	2	1	37.5K
26	80 x 50	640 x 480	4	2	75K
27	80 x 50	640 x 480	16	4	150K
28	80 x 50	640 x 480	256	8	300K

The following mode is available if the configured monitor type is 2 and only on some machines. These monitors may only display this mode. The refresh rate is 64.4Hz and the line rate is 61.2kHz. In this mode, VDU 5 text is single height, while VDU 4 text is double height.

Mode	Text col x row	Resolution hor x ver	Logical colours	Bits per pixel	Memory used
23	144 x 56	1152 x 896	2	1	126K

Notes on display modes

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 23 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 if possible. 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 size of the screen in a given mode can be determined by reading VDU variables XWindLimit, YWindLimit, XEigFactor, YEigFactor.

The other banks may be accessed using OS_Bytes 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 ECF origin is set to (0,0)

- The dot pattern for dotted lines is reset to &AAAAAAAA
- The dot pattern repeat length is reset to 8
- The screen is cleared to the current text background colour (ie black).

If there is not enough configured screen RAM for the mode you have selected, and the application workspace area is not in use, then memory is moved out of the application workspace area to the screen area.

The current screen mode may be read using OS_Byte 135.

Example

`VDU 22,7`	Select Teletext mode

VDU 23

Miscellaneous commands

Syntax

VDU 23,<command>,<par1> - <par8>

Parameters

<command>	The command to perform
<par1> - <par8>	the 8 parameters which follow it

Use

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 '|' as shorthand in VDU statements, eg:

Examples

```
VDU 23,0,10|              These two lines have the same effect
VDU 23,0,10,0,0,0,0,0,0,0
```

VDU 23,0

Set the interlace and controls cursor appearance

Syntax

VDU 23,0,<action>,<mode>,0,0,0,0,0,0

Parameters

<action>	Sets which action to perform
<mode>	Defines the mode for a given action

Use

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

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

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

action= 10 mode 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

action= 11 mode defines the end line for the cursor.

The bottom line of the cursor is 7 for 'normal' modes, 9 for standard 'gap' modes, and 19 for mode 7.

Example

`VDU 23,0,8,&81|` Turn the screen interlace on

VDU 23,1

Control the appearance of the cursor

Syntax

VDU 23,1,<mode>,0,0,0,0,0,0,0

Parameters

<mode>	determines which cursor mode

Use

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

mode	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.

Example

`VDU 23,1,3|` makes the cursor flash

VDU 23,2-5

Define ECF pattern and colours

Syntax

VDU 23,<2|3|4|5>,<n1>,<n2>,<n3>,<n4>,<n5>,<n6>,<n7>,<n8>

Parameters

<n1>,<n2>,<n3>,<n4>,<n5>,<n6>,<n7>,<n8>	colour for each row

Use

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 the colours of 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 BBC/Master compatible mode. The other is the native RISC OS mode which decodes the values in a simpler fashion. To change between these modes use VDU 23,17,4.

If the bit settings in one of the n parameters is denoted by 76543210, then the logical colours of the pixels in each row (from left to right) are:

Bits per pixel	No. of colours	No. of pixels in a line	BBC/Master colours	RISC OS colours
1	2	8	7,6,5,4,3,2,1,0	0,1,2,3,4,5,6,7
2	4	4	73, 62, 51, 40	10, 32, 54, 76
4	16	2	7531, 6420	3210, 7654
8	256	1	76543210	76543210

There are many examples of using these and the VDU 23,12-15 commands to alter ECF functions in the application notes section at the end of this chapter.

In any 256 colour mode, each parameter refers to the colour of each line. Use the colour byte as described by VDU 19.

VDU 23,6

Set dot-dash line style

Syntax

VDU 23,6,<n1>,<n2>,<n3>,<n4>,<n5>,<n6>,<n7>,<n8>

Parameters

<n1>,<n2>,<n3>,<n4>,<n5>,<n6>,<n7>,<n8>	bit pattern for style

Use

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

Example

```
VDU 23,6,&F0,&F0,&F0,&F0,&F0,&F0,&F0,&F0
```

VDU 23,7

Scroll text window or screen

Syntax

VDU 23,7,<extent>,<direction>,<movement>,0,0,0,0,0

Parameters

<extent>	text window or screen
<direction>	direction to scroll
<movement>	how much movement

Use

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

extent	Effect
0	scroll the current text window
1	scroll the entire screen

direction	Effect
0	scroll right
1	scroll left
2	scroll down
3	scroll up
4	scroll in positive X direction
5	scroll in negative X direction
6	scroll in positive Y direction
7	scroll in negative Y direction

movement	Effect
0	scroll by one character cell
1	scroll by one character cell vertically or one byte horizontally

If movement is 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

Example

`VDU 23,7,0,3,0|`	Scroll window up one character

VDU 23,8

Clear a block of the text window

Syntax

VDU 23,8,<base start>,<base end>,<x1>,<y1>,<x2>,<y2>,0,0

Parameters

<base start>	base position of start of block
<base end>	base position of end of block
<x1>,<y1>,<x2>,<y2>	displacements of block

Use

VDU 23,8 causes a block of the current text window to be cleared to the text background colour. The parameters base start and base end 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 the base start and base end; they determine the start and end of the block:

x1	Displacement from base start in x direction
y1	Displacement from base start in y direction
x2	Displacement from base end in x direction
y2	Displacement from base end 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).

Example

`VDU 23,8,5,10,0,0,0,0|` Clear from cursor to end of screen

VDU 23,9

Set flash time for first flashing colour

Syntax

VDU 23,9,<duration>,0,0,0,0,0,0,0

Parameters

<duration>	number of Vsyncs

Use

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

n = 0	sets a steady flash colour 1
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 - 17 it is approximately 1/50th second.

This command is equivalent to OS_Byte 9.

Example

`VDU 23,9,1|`	Set to one Vsync

VDU 23,10

Set flash time for second flashing colour

Syntax	VDU 23,10,<duration>,0,0,0,0,0,0,0
Parameters	<duration> number of Vsyncs
Use	VDU 23,10 sets the flash time for the second flashing colour. The length is determined by the value of duration as follows:

n = 0	sets a steady flash colour 2
n <> 0	sets the duration to n Vsyncs

This command is equivalent to OS_Byte 10.

Example

`VDU 23,10,2|`	Set to two Vsyncs

VDU 23,11

Set default patterns

Syntax

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

Parameters

—

Use

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:

Mode 0

1 – Red-orange	2 – Orange	3 – Yel-orange	4 – Cream
11001100	11110000	11111111	00000011
00000000	00001111	00110011	00001100
11001100	11110000	11111111	01000100
00000000	00110011	01010101	10001000

Modes 1,5,8,11,19,26

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 2,9,12,14,16,17,20,27

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

Modes 4,18,23,25

1 – Dark grey	2 – Grey	3 – Light grey	4 – Hatching
10101010	11001100	11111111	00010001
00000000	00110011	01010101	00100010
10101010	11001100	11111111	01000100
00000000	00110011	01010101	10001000

Modes 10,13,15,21,24,28

1 – Grey		2 – Slate		3 – Green		4 – Pink	
3F	00	0	C0	4	C0	3B	00
	40		80		80		40
	80		40		40		80
	C0		00		00		C0

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

Example

```
VDU 23,11|
```

VDU 23,12-15

Define simple ECF patterns and colours

Syntax

VDU 23,<12 | 13 | 14 | 15>,<n1>,<n2>,<n3>,<n4>,<n5>,<n6>,<n7>,<n8>

Parameters

Define a two by four block of pixels as follows:

n1	n2
n3	n4
n5	n6
n7	n8

Use

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.

In any 256 colour mode, the declared use of the parameters does not apply. In this case, each parameter refers to the colour of each line, from 1 to 8. Use the colour byte as described by VDU 19.

Example

To set up the following pattern in mode 1 for colour pattern 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

Control the movement of cursor after printing

Syntax

VDU 23,16,<x>,<y>,0,0,0,0,0,0

Parameters

<x>	exclusive OR value
<y>	AND value

Use

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) exclusive OR x

The interpretation of the flags is as follows:

Bit	Value	Effect
7	0	Normal.
	1	Undefined.
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.
	1	In VDU 5 mode, cursor movements beyond the edge of the window do not cause special actions. This is the most useful mode of VDU 5; used in the Window Manager.
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.
	1	Cursor does not move after the character is printed.
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.
	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	0	X direction is horizontal, Y direction is vertical.
	1	X direction is vertical, Y direction is horizontal.
2	0	Positive vertical direction is down.
	1	Positive vertical direction is up.
1	0	Positive horizontal direction is right.
	1	Positive horizontal direction is left.
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.
	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 character. For example VDU 127 would cancel it; VDU 9 would execute it.

Example

```
VDU 23,16,%00000100,%11111011|          Set vertical direction down
```

VDU 23,17,0-3

Set the tint for a colour

Syntax

VDU 23,17,<0|1|2|3>,<action>,<tint>,0,0,0,0,0

Parameters

<action>	determines which colour is to be set
<tint>	what the tint is to be set to

Use

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

action	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

This command controls the top two bits of blue, green and red independently of each other, and also 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 tint:

tint	Tint effect
&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 moved to bits 2 - 7 of the colour byte, and their order is changed; the appropriate tint value is shifted down by six bits, into bits 0 and 1, and the two parts are then combined.

Example

`VDU 23,17,0,&C0|`	Set the text foreground colour to lightest tint

VDU 23,17,4

Choose the patterns used to interpret subsequent VDU 23,2 - 5… calls

Syntax	VDU 23,17,4,<patterns>,0,0,0,0,0,0
Parameters	<patterns> which mode of patterns
Use	This command chooses which set of colour patterns are used to interpret subsequent VDU 23,2 - 5… calls, depending on the value of <patterns>:

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

Example

```
VDU 23,17,4,1|
```

Use native colour patterns

VDU 23,17,5

Exchange text foreground and background colours

Syntax

VDU 23,17,5,0,0,0,0,0,0,0

Parameters

—

Use

This command exchanges the current text foreground and background colours. After the first time it's called, subsequent characters printed are in inverse video. After the second time it's called, subsequent characters printed are of normal appearance.

Example

```
VDU 23,17,5|
```

VDU 23,17,6

Set ECF origin

Syntax

VDU 23,17,6,<x>;<y>;0,0,0

Parameters

<x>,<y>;	point coordinates

Use

By default, the alignment of ECF patterns is with the bottom left corner of the screen: This command changes it so that the bottom left of the pattern coincides with the bottom left of the specified point.

The origin is restored to the default after a mode change.

OS_SetECFOrigin (SWI &56) performs the same action.

Example

```
VDU 23,17,6,200;300;0,0,0
```

VDU 23,17,7

Set character size/spacing

Syntax

VDU 23,17,7,<flags>,<x>;<y>;0,0

Parameters

<flags>	what to set the size of
<x>,<y>;	size in pixels

Use

This command allows changing the size and spacing of VDU 5 characters. They are reset when a mode change occurs. flags bit 1 refers to the size of VDU 5 characters. Bit 2 refers to the spacing between VDU 5 characters. x and y are sizes in pixels.

Sizes of 8x16 and 8x8 are optimised for speed. All other settings are much slower. The spacing settings do not affect the speed. The default size and spacing of VDU 5 characters is 8x8.

Example

`VDU 23,17,7,%100,10;8;0,0` change VDU 5 spacing to 10 pixels

VDU 23,18-24

Reserved for future expansion

VDU 23,25-26

These calls are provided by the Font Manager. See the chapter on that for details.

VDU 23,27

This call is provided by the Sprite Manager. See the chapter on this for details.

VDU 23,28-31

Reserved for use by application programs.

VDU 23,32-255

Redefine the printable characters

Syntax

VDU 23,<32-255>,<n1>,<n2>,<n3>,<n4>,<n5>,<n6>,<n7>,<n8>

Parameters

<32 - 255>	character to define
<n1>,<n2>,<n3>,<n4>,<n5>,<n6>,<n7>,<n8>	definition by row

Use

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 value 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 '0' 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 10.

Example

`VDU 23,65,&AA,&55,&AA,&55,&AA,&55,&AA,&55`	redefine 'A'

VDU 24

Define graphics window

Syntax

VDU 24,<x1>;<y1>;<x2>;<y2>;

Parameters

<x1>;<y1>;<x2>;<y2>	coordinates of window

Use

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.

The coordinates are inclusive – that is, the points you specify lie within the window.

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

Example

```
VDU 24,100;150;700;800;
```

The following example shows how to derive (in this instance, xsize) the size of a window in OS units

```
DIM blk% 12
VduExt_XEigFactor% = 4
VduExt_XWindLimit% = 11
blk%!0 = VduExt_XEigFactor%
blk%!4 = VduExt_XWindLimit%
blk%!8 = -1
SYS "OS_ReadVduVariables", blk%, blk%
xeigfactor% = blk%!0
xwindlimit% = blk%!4: REM in pixels
xwindsize%  = (xwindlimit% + 1) << xeigfactor%: REM in OS units
```

VDU 25

General PLOT command

Syntax

VDU 25,<type>,<x>;<y>;

Parameters

<type>	what kind of plot to perform
<x>;<y>;	where to plot

Use

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

The bottom three bits of type determine the manner in which the plot is to be performed. Thus (type AND 7) has the following effects:

type AND 7	Effect
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 coordinates 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 type determine the action to be performed. The value given here is added to the 0 - 7 range above to get all the possible combinations. The value here is the decimal starting value:

Value	Effect
0	Solid line including both end points
8	Solid line excluding the final point
16	Dotted line including both endpoints, pattern restarted
24	Dotted line excluding the final point, pattern restarted
32	Solid line excluding the initial point
40	Solid line excluding both end points
48	Dotted line excluding the initial point, pattern continued
56	Dotted line excluding both end points, pattern continued

64	Point Plot
72	Horizontal line fill (left and right) to non-background
80	Triangle fill
88	Horizontal line fill (right only) to background
96	Rectangle fill
104	Horizontal line fill (left and right) to foreground
112	Parallelogram fill
120	Horizontal line fill (right only) to non-foreground
128	Flood to non-background
136	Flood to foreground
144	Circle outline
152	Circle fill
160	Circular arc
168	Segment
176	Sector
184	Block copy/move *
192	Ellipse outline
200	Ellipse fill
208	Font printing – see the chapter entitled *The Font manager*
216	Reserved for Acorn Expansion
224	Reserved for Acorn Expansion
232	Sprite Plot – see the chapter on sprites
240	Reserved for User programs
248	Reserved for User programs

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

Value	**Effect**
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, and is opposite the second one.
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 coordinate. Plot ellipse to highest or lowest point on the ellipse.

Example

`VDU 25,69,100;200;` plot point absolute

VDU 26

Restore default windows

Syntax

VDU 26

Parameters

—

Use

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

Syntax	VDU 27
Parameters	—
Use	This VDU has no effect

VDU 28

Define text window

Syntax

VDU 28,<x1>,<y1>,<x2>,<y2>

Parameters

<x1 >	left-most x column
<y1>	bottom-most y row
<x2 >	right-most x column
<y2 >	top-most y row

Use

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

If the command attempts to define a window which extends outside the screen boundaries, has x1 greater than x2, or has y1 less than y2, 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.

Example

```
VDU 28,10,15,20,5
```

VDU 29

Set graphics origin

Syntax

VDU 29,<x>;<y>;

Parameters

<x>,<y>	where the origin is to be set

Use

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 coordinates of the new origin.

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

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

Example

```
VDU 29,100;200;
```

VDU 30

Home text cursor

Syntax

VDU 30

Parameters

—

Use

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 (character size –1) 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

Positior. text cursor

Syntax

VDU 31,<x>,<y>

Parameters

<x>,<y>	text position to move to

Use

VDU 31 moves the text cursor to a specified x and y coordinate 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 coordinate 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 (x character spacing * x) pixels in the positive X direction, plus (y character spacing * 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 134.

Example

```
VDU 31,10,5
```

VDU 127

Delete

Syntax

VDU 127

Parameters

—

Use

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.

OS_Byte 9 (SWI &06)

Write duration of first flash colour

On entry

R0 = 9 (reason code)
R1 = new duration to write

On exit

R0 = preserved
R1 = duration before being overwritten
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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 for 50Hz screen modes 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 195.

Related SWIs

OS_Byte 10 (SWI &06), OS_Byte 195 (SWI &06)

Related vectors

ByteV

OS_Byte 10 (SWI &06)

Write duration of second flash colour

On entry

R0 = 10
R1 = duration to write

On exit

R0 = preserved
R1 = duration before being overwritten
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the duration for the second flash colour. See OS_Byte 9 for an explanation.

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

Related SWIs

OS_Byte 9 (SWI &06), OS_Byte 194 (SWI &06)

Related vectors

ByteV

OS_Byte 19 (SWI &06)

Wait for vertical sync

On entry

R0 = 19

On exit

R0 = preserved
R1 = corrupted
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

From this time until the next frame starts to be displayed (≈3.5ms for modes 0–17 and 24, ≈0.8ms for modes 18–21 and 23, and ≈1.4ms for modes 25-28), you have this time to redraw the screen.

It is possible to have more than this time by drawing from top to bottom, or setting a timer to wait until video output has passed the place on the screen you want to redraw.

If even this 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_Bytes 112–113 for example).

Related SWIs

OS_Byte 112 (SWI &06), OS_Byte 113 (SWI &06)

Related vectors

ByteV

OS_Byte 20 (SWI &06)

Reset font definitions

On entry	R0 = 20

On exit	R0 = preserved
	R1 = corrupted
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	The shape of the character displayed when printing ASCII codes 32–255 may be redefined using the VDU 23,32–255 commands. 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 25 for details on how to restore the other codes or how to restore a smaller selected group.
Related SWIs	OS_Byte 25 (SWI &06)
Related vectors	ByteV

OS_Byte 25 (SWI &06)

Reset group of font definitions

On entry

R0 = 25
R1 = group to restore

On exit

R0 = preserved
R1 = corrupted
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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	Range of characters to restore
0	32–255
1	32–63
2	64–95
3	96–127
4	128–159
5	160–191
6	192–223
7	224–255

Related SWIs

OS_Byte 20 (SWI &06)

Related vectors

ByteV

OS_Byte 106 (SWI &06)

Select pointer/activate mouse

On entry	R0 = 106 R1 = pointer shape and linkage flag

On exit	R0 = preserved
	R1 = shape and linkage flag before being overwritten
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined

Use

You can define four 'pointer buffers' using OS_Word 21; each holding a different shape definition for the mouse pointer. This call allows you to select one of these definitions for future use, or to turn off the pointer depending on the bottom seven 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**
&00	Link pointer to mouse
&80	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.

Related SWIs

OS_Word 21 (SWI &07)

Related vectors

ByteV

OS_Byte 112 (SWI &06)

Write VDU driver screen bank

On entry	R0 = 112
	R1 = bank number

On exit	R0 = preserved
	R1 = previous bank number
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined

Use

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 (TotalScreenSize)/(ModeSize), where TotalScreenSize is the amount actually present in screen memory 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 250 may be used to read the bank number without writing it

Related SWIs

OS_Byte 250 (SWI &06)

Related vectors

ByteV

OS_Byte 113 (SWI &06)

Write display hardware screen bank

On entry	R0 = 113
	R1 = bank number

On exit	R0 = preserved
	R1 = value before being overwritten
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	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 251.

Related SWIs	OS_Byte 251 (SWI &06)
Related vectors	ByteV

OS_Byte 114 (SWI &06)

Write shadow/non-shadow state

On entry	R0 = 114
	R1 = shadow state

On exit	R0 = preserved
	R1 = value before being overwritten
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	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 112 and 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.

Related SWIs	OS_Byte 112 (SWI &06), OS_Byte 113 (SWI &06)
Related vectors	ByteV

OS_Byte 117 (SWI &06)

Read VDU status

On entry	R0 = 117

On exit	R0 = preserved
	R1 = status byte

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	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

Related SWIs	None
Related vectors	ByteV

OS_Byte 134 (SWI &06)

Read text cursor position

On entry

R0 = 134

On exit

R0 = preserved
R1 = position in x direction
R2 = position in y direction

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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 165 reads the position of the output cursor irrespective of cursor editing mode.

Text is printed at x 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.

Related SWIs

OS_Byte 165 (SWI &06)

Related vectors

ByteV

OS_Byte 135 (SWI &06)

Read character at text cursor position and screen mode

On entry

R0 = 135

On exit

R0 = preserved
R1 = ASCII value of character (0 if unreadable)
R2 = screen mode

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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 as input the next time Copy is pressed).

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

Related SWIs

None

Related vectors

ByteV

OS_Byte 144 (SWI &06)

Set vertical screen shift and interlace

On entry

R0 = 144
R1 = vertical screen shift (as a signed 8 bit number)
R2 = interlace flag

On exit

R0 = preserved
R1 = previous vertical screen shift
R2 = previous interlace flag

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

This is equivalent to *TV, which is described in this chapter.

Related SWIs

None

Related vectors

ByteV

OS_Byte 160 (SWI &06)

Read VDU variable value

On entry

R0 = 160
R1 = VDU variable number (0–15)

On exit

R0 = preserved
R1 = value of variable
R2 = value of next variable (R1 on entry + 1)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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 RISC OS.

Value	Location
0	LSB of graphics window left column (ic)
1	MSB of graphics window left column (ic)
2	LSB of graphics window bottom row (ic)
3	MSB of graphics window bottom row (ic)
4	LSB of graphics window right column (ic)
5	MSB of graphics window right column (ic)
6	LSB of graphics window top row (ic)
7	MSB of graphics window top row (ic)
8	Text window left column
9	Text window bottom row
10	Text window right column
11	Text window top row
12	LSB of graphics origin X coordinate (ec)

13	MSB of graphics origin X coordinate (ec)
14	LSB of graphics origin Y coordinate (ec)
15	MSB of graphics origin Y coordinate (ec)

- (ic) means internal coordinates: the origin is always the bottom left of the screen. One unit is one pixel wide and one pixel high.
- (ec) means external coordinates: a pixel is (1 « XEigFactor) units wide and (1 « YEigFactor) units high, where XEigFactor and YEigFactor are VDU variables.

This OS_Byte is provided mainly for compatibility with the BBC/Master 128. You can read many more of the VDU variables using OS_ReadVduVariables and OS_ReadModeVariable.

Related SWIs

OS_ReadVduVariables (SWI &31), OS_ReadModeVariable (SWI &35)

Related vectors

ByteV

OS_Byte 163 (SWI &06)

Read/write general graphics information

On entry

R0 = 163
R1 = 242
R2 = dot-dash repeat length or action code

On exit

R0 = preserved
R1 = preserved or status
R2 = preserved or status

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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 (0 means 64)

R2 bits	Meaning
Bits 0–31	Current size of the system sprite area in bytes.

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

R1 = width in pixels (ie. internal coordinates)
R2 = height in pixels (ie. internal coordinates)

Related SWIs

None

Related vectors

ByteV

OS_Byte 165 (SWI &06)

Read output cursor position

On entry	R0 = 165
On exit	R0 = preserved R1 = position in x direction R2 = position in y direction
Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call returns the position of the output cursor, even while cursor editing is in progress.
Related SWIs	None
Related vectors	ByteV

OS_Byte 193 (SWI &06)

Read/write flash counter

On entry	R0 = 193
	R1 = 0 to read or new duration to write
	R2 = 255 to read or 0 to write

On exit	R0 = preserved
	R1 = duration before being overwritten
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	The duration stored is changed by being masked with R2 and then exclusive ORd with R1. ie. ((duration AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits. 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, providing that the current flash period is not infinite. When it reaches zero, the colours are swapped and the counter is loaded with the duration of the second colour.
Related SWIs	None
Related vectors	ByteV

OS_Byte 194 (SWI &06)

Read duration of second colour

On entry	R0 = 194
	R1 = 0
	R2 = 255

On exit	R0 = preserved
	R1 = duration
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This command will read the location that has been set by OS_Byte 10. This must not be used to write this location, as RISC OS would then not match the location value. This call is only included for compatibility.
Related SWIs	OS_Byte 10 (SWI &06)
Related vectors	ByteV

OS_Byte 195 (SWI &06)

Read duration of first colour

On entry	R0 = 195
	R1 = 0
	R2 = 255

On exit	R0 = preserved
	R1 = duration
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This command will read the location that has been set by OS_Byte 9. This must not be used to write this location, as RISC OS would then not match the location value. This call is only included for compatibility.
Related SWIs	OS_Byte 9 (SWI &06)
Related vectors	ByteV

OS_Byte 211 (SWI &06)

Read/write bell channel

On entry	R0 = 211
	R1 = 0 to read or new channel to write
	R2 = 255 to read or 0 to write

On exit	R0 = preserved
	R1 = channel before being overwritten
	R2 = bell sound information (see OS_Byte 212)

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	The channel stored is changed by being masked with R2 and then exclusive ORd with R1. ie. ((channel AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits. 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.
Related SWIs	OS_Byte 212 (SWI &06), OS_Byte 213 (SWI &06) OS_Byte 214 (SWI &06)
Related vectors	ByteV

OS_Byte 212 (SWI &06)

Read/write bell volume

On entry	R0 = 212
	R1 = 0 to read or new volume to write
	R2 = 255 to read or 0 to write

On exit	R0 = preserved
	R1 = volume before being overwritten
	R2 = bell frequency (see OS_Byte 213)

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	The volume stored is changed by being masked with R2 and then exclusive ORd with R1. ie. ((volume AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits. This allows you to read or set the volume of the sound used to make the Ctrl-G 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).
Related SWIs	OS_Byte 211 (SWI &06), OS_Byte 213 (SWI &06), OS_Byte 214 (SWI &06)
Related vectors	ByteV

OS_Byte 213 (SWI &06)

Read/write bell frequency

On entry	R0 = 213
	R1 = 0 to read or new frequency to write
	R2 = 255 to read or 0 to write

On exit	R0 = preserved
	R1 = frequency before being overwritten
	R2 = bell duration (see OS_Byte 214)

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	The frequency stored is changed by being masked with R2 and then exclusive ORd with R1. ie. ((frequency AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits. 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 *Sound command. Note that all frequencies are provided for compatibility only; new RISC OS values cannot be used.
Related SWIs	OS_Byte 211 (SWI &06), OS_Byte 212 (SWI &06), OS_Byte 214 (SWI &06)
Related vectors	ByteV

OS_Byte 214 (SWI &06)

Read/write bell duration

On entry	R0 = 214
	R1 = 0 to read or new duration to write
	R2 = 255 to read or 0 to write

On exit	R0 = preserved
	R1 = duration before being overwritten
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	The duration stored is changed by being masked with R2 and then exclusive ORd with R1. ie. ((duration AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits. 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.
Related SWIs	OS_Byte 211 (SWI &06), OS_Byte 212 (SWI &06), OS_Byte 213 (SWI &06)
Related vectors	ByteV

OS_Byte 217 (SWI &06)

Read/write paged mode line count

On entry	R0 = 217
	R1 = 0 to read or new count to write
	R2 = 255 to read or 0 to write

On exit	R0 = preserved
	R1 = count before being overwritten
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	The count stored is changed by being masked with R2 and then exclusive ORd with R1. ie. ((count AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits. 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 0 or OS_ReadLine to perform the input, this call is made automatically. OS_Word 0 is provided for compatiblity only and should not be used.
Related SWIs	None
Related vectors	ByteV

OS_Byte 218 (SWI &06)

Read/write bytes in VDU queue

On entry	R0 = 218
	R1 = 0 to read or new count to write
	R2 = 255 to read or 0 to write

On exit	R0 = preserved
	R1 = count before being overwritten
	R2 = corrupted

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	The count stored is changed by being masked with R2 and then exclusive ORd with R1. ie. ((count AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits. 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). You can use this call when an escape condition is acknowledged. This prevents the first few characters of an error message from being 'swallowed' by an incomplete VDU sequence.
Related SWIs	None
Related vectors	ByteV

OS_Byte 250 (SWI &06)

Read VDU driver screen bank number

On entry	R0 = 250
	R1 = 0
	R2 = 255

On exit	R0 = preserved
	R1 = screen bank used by VDU drivers
	R2 = display screen bank (see OS_Byte 251)

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This command will read the location that has been set by OS_Byte 112. This must not be used to write this location, as RISC OS would then not match the location value.
Related SWIs	OS_Byte 112 (SWI &06)
Related vectors	ByteV

OS_Byte 251 (SWI &06)

Read display screen bank number

On entry

R0 = 251
R1 = 0
R2 = 255

On exit

R0 = preserved
R1 = screen bank used by the display
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This command will read the location that has been set by OS_Byte 113.

This must not be used to write this location, as the hardware would then not match the location value.

Related SWIs

OS_Byte 113 (SWI &06)

Related vectors

ByteV

OS_Word 9 (SWI &07)

Read pixel logical colour

On entry

R0 = 9 (reason code)
R1 = pointer to parameter block
  R1+0 = LSB of X coordinate
  R1+1 = MSB of X coordinate
  R1+2 = LSB of Y coordinate
  R1+3 = MSB of Y coordinate

On exit

R0 = preserved
R1 = preserved
  R1+4 = the logical colour of the pixel specified.

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

- the screen is in a 256 colour mode
- 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 (SWI &32) instead. This returns both the logical colour and tint. The OS_Word should be used for compatibility purposes only, since the tint is discarded.

Related SWIs

OS_ReadPoint (SWI &32)

Related vectors

WordV

OS_Word 10 (SWI &07)

Read a character definition

On entry

R0 = 10
R1 = pointer to parameter block
R1+0 = ASCII code of character required

On exit

R0 = preserved
R1 = preserved
    R1+1 = top row of definition
    :
    R1+10 = bottom row of definition

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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 apart from the following characters:

Value	Information returned
2	ECF pattern 1 (in native mode)
3	ECF pattern 2 (in native mode)
4	ECF pattern 3 (in native mode)
5	ECF pattern 4 (in native mode)
6	Dot-dash pattern

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.

Related SWIs	None
Related vectors	WordV

OS_Word 11 (SWI &07)

Read the palette

On entry

R0 = 11
R1 = pointer to parameter block
R1+0 = logical colour to read

On exit

R0 = preserved
R1 = preserved
    R1+1 = physical colour associated with the specified logical colour
    R1+2 = red component
    R1+3 = green component
    R1+4 = blue component

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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, or to read the border and pointer palettes, you should use OS_ReadPalette (SWI &2F). The OS_Word is provided for compatibility only.

Related SWIs

OS_ReadPalette (SWI &2F)

Related vectors

WordV

OS_Word 12 (SWI &07)

Write the palette

On entry

R0 = 12
R1 = pointer to parameter block
  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

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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.

Related SWIs

None

Related vectors

WordV

OS_Word 13 (SWI &07)

Read current and previous graphics cursor positions

On entry	R0 = 13 R1 = pointer to parameter block
On exit	R0 = preserved R1 = preserved R1+0 = LSB of previous X coordinate R1+1 = MSB of previous X coordinate R1+2 = LSB of previous Y coordinate R1+3 = MSB of previous Y coordinate R1+4 = LSB of current X coordinate R1+5 = MSB of current X coordinate R1+6 = LSB of current Y coordinate R1+7 = MSB of current Y coordinate
Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	All the coordinates are in external form. You can read points visited before the previous one (and many other VDU variables) using OS_ReadVduVariables (SWI &31).
Related SWIs	OS_ReadVduVariables (SWI &31)
Related vectors	WordV

OS_Word 21,0 (SWI &07)

Define pointer size, shape and active point

On entry

R0 = 21
R1 = pointer to parameter block

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

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

You can define four shapes. These are numbered one to four and may be selected using OS_Byte 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 coordinates (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.

However, note that colour two should be used with caution in defining pointer shapes, as it does not work correctly on high-res mono screens.

Related SWIs

OS_Byte 106 (SWI &06)

Related vectors

WordV

OS_Word 21,1 (SWI &07)

Define mouse coordinate bounding box

On entry	R0 = 21 R1 = pointer to parameter block R1+0 = 1 (sub-reason code) R1+1 = LSB of left coordinate R1+2 = MSB of left coordinate R1+3 = LSB of bottom coordinate R1+4 = MSB of bottom coordinate R1+5 = LSB of right coordinate R1+6 = MSB of right coordinate R1+7 = LSB of top coordinate R1+8 = MSB of top coordinate	All treated as signed 16-bit values, relative to screen origin at the time the command is issued

On exit	R0 = preserved
	R1 = preserved	

Interrupts	Interrupt status is not altered Fast interrupts are enabled	
Processor Mode	Processor is in SVC mode	
Re-entrancy	Not defined	

Use

The coordinates should be given as signed 16-bit values relative to the graphics 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 buffer is not flushed, but any buffered coordinates will be moved inside the bounding box when they are read

When the mode changes, the box is set to the size of the screen.

Related SWIs

None

Related vectors

WordV

OS_Word 21,2 (SWI &07)

Define mouse multipliers

On entry	R0 = 21 R1 = pointer to parameter block R1+0 = 2 R1+1 = X multiplier (these are both treated as signed 8-bit values) R1+2 = Y multiplier

On exit	R0 = preserved
	R1 = preserved

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	The multipliers control the ratio between the movement of the mouse and the change in the coordinates 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 the configured MouseStep value. This is initially 1, when a movement of approximately 15cm of the mouse will move the pointer across the screen.
Related SWIs	None
Related vectors	WordV

OS_Word 21,3 (SWI &07)

Set mouse position

On entry	R0 = (reason code) R1 = pointer to parameter block 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

On exit	R0 = preserved
	R1 = preserved

Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	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.
Related SWIs	None
Related vectors	WordV

OS_Word 21,4 (SWI &07)

Read unbuffered mouse position

On entry	R0 = 21 R1 = pointer to parameter block R1+0 = 4
On exit	R0 = preserved R1 = preserved R1+1 = LSB of X position R1+2 = MSB of X position R1+3 = LSB of Y position R1+4 = MSB of Y position
Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call will read the position of the mouse at the time of the call. That is, it will not read the position from the mouse buffer. Care should be taken when reading this position, as the buffer positions may be significantly out of step.
Related SWIs	None
Related vectors	WordV

OS_Word 21,5 (SWI &07)

Set pointer position

On entry

R0 = (reason code)
R1 = pointer to parameter block
    R1+0 = 5
    R1+1 = LSB of X position
    R1+2 = MSB of X position
    R1+3 = LSB of Y position
    R1+4 = MSB of Y position

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The new values for the X and Y positions of the pointer are given as two signed 16-bit values.

Note that this call sets the position of the pointer rather than the mouse. If the mouse and pointer are linked, then the pointer position will be updated with the mouse position on the next VSync interrupt.

Related SWIs

None

Related vectors

WordV

OS_Word 21,6 (SWI &07)

Read pointer position

On entry	R0 = 21 R1 = pointer to parameter block R1+0 = 6
On exit	R0 = preserved R1 = preserved R1+1 = LSB of X position R1+2 = MSB of X position R1+3 = LSB of Y position R1+4 = MSB of Y position
Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call will read the position of the pointer. If the mouse and pointer are not linked, then this call read the position that the pointer was last set to. If they are linked, then the pointer is updated from the unbuffered mouse position every Vsync; otherwise 9 clicks while the Desktop is busy would freeze the pointer.
Related SWIs	None
Related vectors	WordV

OS_Word 22 (SWI &07)

Write screen base address

On entry

R0 = 22
R1 = pointer to parameter block
R1+0 = Type
R1+1 = Least significant byte of offset
R1+2 ...
R1+3 ...
R1+4 = Most significant byte of offset

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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 configured ScreenSize determines the amount of RAM initially set aside for the screen. This can subsequently change, for example if you drag the *screen memory* bar in the Task Manager, or call OS_ChangeDynamicArea. You can read the current amount set aside for the screen by reading the VDU variable TotalScreenSize; and you can read the amount needed for a single screen by reading the mode variable ScreenSize.

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

Related SWIs

OS_Byte 112 (SWI &06), OS_Byte 113 (SWI &06)

Related vectors

WordV

OS_Mouse (SWI &1C)

Read a mouse state from the buffer

On entry

On exit

R0 = mouse X coordinate
R1 = mouse Y coordinate
R2 = mouse buttons
R3 = time of button change

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered as interrupts are disabled

Use

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 coordinates will lie within the mouse bounding box, which initially defaults to the screen area. 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).

Related SWIs

OS_ReadMonotonicTime (SWI &42)

Related vectors

MouseV

OS_ReadPalette (SWI &2F)

Read the palette setting of a colour

On entry

R0 = logical colour
R1 = type of colour

On exit

R2 = setting of first flashing colour
R3 = setting of second flashing colour

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_ReadPalette reads the setting of a particular colour that is sent to the hardware. R1 selects whether the normal colour, border colour or pointer colour is read as follows:

Value	**Meaning**
16	Read normal colour
24	Read border colour
25	Read pointer 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**
0–6	Value showing how colour was programmed
7	Supremacy bit
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 (bit 7 is the supremacy bit). 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

Related SWIs

None

Related vectors

None

OS_ReadVduVariables (SWI &31)

Read a series of VDU variables

On entry

R0 = pointer to input block
R1 = pointer to output block

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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:

Name	Number	Meaning
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 coordinate of the graphics origin (ec)
OrgY	137	Y coordinate of the graphics origin (ec)

GCsX	138	X coordinate of the graphics cursor (ec)
GCsY	139	Y coordinate of the graphics cursor (ec)
OlderCsX	140	X coordinate of oldest graphics cursor (ic)
OlderCsY	141	Y coordinate of oldest graphics cursor (ic)
OldCsX	142	X coordinate of previous graphics cursor (ic)
OldCsY	143	Y coordinate of previous graphics cursor (ic)
GCsIX	144	X coordinate of graphics cursor (ic)
GCsIY	145	Y coordinate of graphics cursor (ic)
NewPtX	146	X coordinate of new point (ic)
NewPtY	147	Y coordinate 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	Amount of memory currently allocated to the screen
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
GCharSizeX	162	X size of VDU 5 chars (in pixels)
GCharSizeY	163	Y size of VDU 5 chars (in pixels)
GCharSpaceX	164	X spacing of VDU 5 chars (in pixels)
GCharSpaceY	165	Y spacing of VDU 5 chars (in pixels)
HLineAddr	166	Address of fast line-draw routine
TCharSizeX	167	X size of VDU 4 chars (in pixels)
TCharSizeY	168	Y size of VDU 4 chars (in pixels)
TCharSpaceX	169	X spacing of VDU 4 chars (in pixels)
TCharSpaceY	170	Y spacing of VDU 4 chars (in pixels)
GcolOraEorAddr	171	Address of colour blocks for current GCOLs
WindowWidth	256	Characters that will fit on a row of the text window without a newline being generated
WindowHeight	257	Rows that will fit in the text window without scrolling it

- *ic* means internal coordinates, where (0,0) is always the bottom left of the screen. One unit is one pixel.
- *ec* means external coordinates, where (0,0) means the graphics origin, and the size of one unit depends on the resolution. The number of external units on a screen is dependent upon the video mode used; for example MODE 16 has 1280 by 1024 external units. The graphics origin is stored in external coordinate units, but is relative to the bottom left of the screen.
- new point is the internal form of the coordinates given in an unrecognised PLOT command. When the UKPlot vector is called, the internal format coordinates (variables 140–145) have not yet been shuffled down, so the graphics cursor (144–5) contains the coordinates of the last point visited. The external coordinates version of the current point (138–9) is updated from the coordinate given in the unrecognised plot.
- *HLineAddr* points to a fast horizontal line draw routine. It is called as follows:

  R0 = left x-coordinate of end of line
  R1 = y-coordinate of line
  R2 = right x-coordinate of end of line
  R3 = 0 = plot with no action (ie. do nothing)
  1 = plot using foreground colour and action
  2 = invert current screen colour
  3 = plot using background colour and action
  >=4 = pointer to colour block (on 64-byte boundary)

Offset	Value
0	OR mask for top ECF line
4	exclusive OR mask for top ECF line
8	OR mask for next ECF line
12	exclusive OR mask for next ECF line
:	
56	OR mask for bottom ECF line
60	exclusive OR mask for bottom ECF line

  R14 = return address
  Must be entered in SVC mode
  All registers are preserved on exit

  All coordinates are in terms of pixels from the bottom left of the screen. The line is clipped to the graphics window, and is plotted using the colour action specified by R3. The caller must have previously called OS_RemoveCursors and call OS_RestoreCursors afterwards.

- GcolOraEorAddr points to colour blocks for current GCOLs. If the value returned is n, then:
  n+&00–n+&3F is a colour block for the foreground colour + action
  n+&40–n+&7F is a colour block for the background colour + action
  n+&80–n+&BF is a colour block for the background colour
  with store action

  Each colour block is as described above. These are updated whenever a GCOL or TINT is issued or the ECF origin is changed. They are intended for programs which want to access screen memory directly and have access to the current colour/action settings.

Related SWIs

OS_ReadModeVariable (SWI &35)

Related vectors

None

OS_ReadPoint (SWI &32)

Read the colour of a point

On entry	R0 = X coordinate
	R1 = Y coordinate

On exit	R0 = preserved
	R1 = preserved
	R2 = colour
	R3 = tint
	R4 = screen flag

Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant

Use

The coordinates passed 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)

See VDU 19 for a description of colour and tint values.

Related SWIs	None
Related vectors	None

OS_ReadModeVariable (SWI &35)

Read information about a screen mode

On entry

R0 = screen mode, or –1 for current mode
R1 = variable number

On exit

R0 = preserved
R1 = preserved
R2 = value of variable
the C flag is set if variable or mode numbers were invalid

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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:

Name	**Number**		**Meaning**
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
	Bit 3	= 0	non-gap mode
		= 1	'BBC' gap mode (eg modes 3 and 6)
	Bit 4	= 0	not Hi Res mono mode
		= 1	Hi Res mono mode
	Bit 5	= 0	VDU characters are normal height

		= 1	VDU characters are double height
	Bit 6	= 0	harware scroll used
		= 1	hardware scroll never used
ScrRCol	1	Maximum column number for printing text ie. number of columns–1	
ScrBRow	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 number of bits by which an X–coordinate must be shifted right to convert to screen pixels. Thus if this value is n, then one screen pixel corresponds to 2^n external coordinates in the X –direction.	
YEigFactor	5	This indicates the number of bits by which a Y–coordinate must be shifted right to convert to screen pixels. Thus if this value is n, then one screen pixel corresponds to 2^n external coordinates in the Y –direction.	
LineLength	6	Number of bytes on a pixel row This is the same as (characters per row)*(bits per pixel)*(pixel width of character)/8. For example, in mode 15 it is 80*8*8/8, or 640.	
ScreenSize	7	Number of bytes one screen buffer occupies. This must be a multiple of 256 bytes.	
YShftFactor	8	Scaling factor for start address of a screen row. This variable is kept for compatibility reasons and should not be used.	
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.

XWindLimit	11	Number of X pixels on screen–1
YWindLimit	12	Number of Y pixels on screen–1

Related SWIs

OS_ReadVduVariables (SWI &31)

Related vectors

None

OS_RemoveCursors (SWI &36)

Remove the cursors from the screen

On entry

–

On exit

–

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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.

This call is provided only for routines that need direct screen access.

Note that routines that directly access the screen may need to run in SVC mode if the routines are to work with hardware scrolled screens, which may straddle the logical-physical memory boundary at 32MByte. If the routines do not need to work with hardware scrolled screens, then USR mode is adequate..

Related SWIs

OS_RestoreCursors (SWI &37)

Related vectors

None

OS_RestoreCursors (SWI &37)

Restore the cursors to the screen

On entry	–
On exit	–
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	OS_RestoreCursors restores the cursor state previously saved on the internal stack using OS_RemoveCursors. This call is provided only for routines that need direct screen access.
Related SWIs	OS_RemoveCursors (SWI &36)
Related vectors	None

OS_CheckModeValid (SWI &3F)

Check if it is possible to change to a specified mode

On entry	R0 = mode number to check
On exit	if C flag = 0 then mode is valid R0 = preserved if C flag = 1 then mode is invalid R0 = –1 if mode is non-existent R0 = –2 if not enough memory R1 = mode that would be used R1 = –2 if unable to select alternative mode
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	OS_CheckModeValid determines whether you can change to a given mode and return with the appropriate carry set. If the mode you are checking isn't available on the current type of monitor, then R1 will contain the mode that will be used if an attempt is made to select the mode which you are checking, using VDU 22. If there is insufficient memory or the call is unable to determine an alternative for another reason, then –2 will be returned. If this call returns that there is insufficient memory for the required mode, then it can be borrowed from other areas of the machine. See the chapter on memory management for details.
Related SWIs	None
Related vectors	None

OS_Plot (SWI &45)

Direct VDU call

On entry	R0 = plot command code
	R1 = x coordinate
	R2 = y coordinate

On exit	R0, R1, R2 = corrupted
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	This call is equivalent to a VDU 25 command. However, it is much more efficient as only one call is required (instead of six calls to OS_WriteC). The call goes directly to the VDU drivers unless spooling has been turned on, redirection has been turned on or if WrchV has been claimed.
Related SWIs	None
Related vectors	WrchV

OS_SetECFOrigin (SWI &56)

Set the origin of the ECF patterns

On entry	R0 = x coordinate
	R1 = y coordinate

On exit	R0 = preserved
	R1 = preserved

Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	By default, the alignment of ECF patterns is with the bottom left corner of the screen. This command makes the bottom left of the pattern coincide with the bottom left of the specified point. The origin is restored to the default after a mode change. VDU 23,17,6 performs the same action.
Related SWIs	None
Related vectors	None

OS_ReadSysInfo (SWI &58)

Read screen size after hard reset

On entry	R0 = 0
On exit	R0 = size in bytes
Interrupts	Interrupt status is not altered Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is re-entrant
Use	This call will return the screen size in bytes after the next hard reset.
Related SWIs	None
Related vectors	None

OS_ChangedBox (SWI &5A)

Determine which area of the screen has changed

On entry

R0 = 0 – disable changed box calculations
1 – enable changed box calculations
2 – reset changed box to null rectangle
–1 – read changed box information

On exit

R0 = previous enable state in bit 0 (0 – disabled, 1 – enabled)
R1 points to a fixed block of 5 words, containing the following info
R1+0 = new disable/enable flag (in bit 0)
R1+4 = x-coordinate of left edge of box
R1+8 = y-coordinate of bottom edge of box
R1+12 = x-coordinate of right edge of box
R1+16 = y-coordinate of top edge of box

The (R1+4) to (R1+16) values are only valid if the change box calculations were in an enabled state immediately after the call, otherwise they are undefined.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call checks which areas of the screen have changed over calls to the VDU drivers. When this feature is enabled, RISC OS maintains the coordinates of a rectangle which completely encloses any areas that have changed since the last time the rectangle was reset.

This is particularly useful for applications which switch output to sprites, and then want to repaint the sprite onto the screen after performing VDU operations on the sprite. The application can make significant speed improvements by only repainting the section of the sprite which corresponds to the changed box.

All coordinates are measured in pixels from the bottom left of the screen. If a module provides extensions to the VDU drivers, it should read the address of this block on initialisation, and update the coordinates as appropriate. If an exact calculation of which areas have been modified is difficult, then the module should extend the rectangle to include the whole of the graphics window (or indeed the whole screen, if the operation can affect areas outside the graphics window).

The disable/enable flag at offset 0 in the block is for information only – it must not be modified directly, as RISC OS holds the master copy of this flag.

Changed box calculations are disabled on a mode change. However, the disable/enable state and the coordinates of the rectangle form part of the information held in save areas when output is switched between the screen and sprites.

Related SWIs

None

Related vectors

None

*Configure Loud

Sets the beep at full volume

Syntax

```
*Configure Loud
```

Parameters

None

Use

This command sets the configured volume for the bell to its loudest volume. The change takes effect on the next reset.

Related commands

*Configure Quiet

Related SWIs

OS_Byte 212 (SWI &06)

Related vectors

None

*Configure Mode

Selects the screen mode used after a reset

Syntax	`*Configure Mode <n>`
Parameters	`<n>` screen mode number
Use	This command selects the screen mode for use after a hard reset, or on leaving the desktop. Mode 0 is the default.
Example	`*Configure Mode 27` selects VGA mode with 16 colours
Related commands	VDU 22
Related SWIs	None
Related vectors	None

*Configure MonitorType

Selects the default monitor type

Syntax

```
*Configure MonitorType <n>
```

Parameters

`<n>`	0 to 3

Use

This specifies the kind of monitor that is connected to the computer, as follows:

Type	Monitor
0	50Hz TV standard colour monitor
1	Multiscan monitor
2	Hi-resolution 64Hz monochrome monitor
3	60Hz VGA-type monitor

The monitor type can also be configured by holding down the corresponding key from the numeric keypad while the computer is switched on.

Example

```
*Configure MonitorType 3
```

Related commands

VDU 22

Related SWIs

None

Related vectors

None

*Configure MouseStep

Selects how fast the pointer moves as you move the mouse

Syntax

```
*Configure MouseStep <n>
```

Parameters

`<n>`	a number between 1 and 127

Use

Useful values for this command are 1, 2 or 3 for slow, medium or fast, respectively.

The mouse position is moved by <n> coordinates for each movement of the mouse. Although values up to 127 are accepted, anything above 6 is impractical because the step is too large.

This can also be set from the desktop, using the Configure application. OS_Word 21,2 can also be used to dynamically set mouse multipliers.

Example

`*Configure MouseStep 3`	select a fast speed

Related commands

None

Related SWIs

None

Related vectors

None

*Configure NoScroll

Sets the screen not to scroll upwards at the end of a line

Syntax

```
*Configure NoScroll
```

Parameters

None

Use

This prevents a newline from being generated when a character is printed at the end of a line. The default is Scroll.

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 character. For example VDU 127 would cancel it; VDU 9 would execute it.

Refer to VDU 23,16 for a lengthier description of noscroll and to show how to dynamically set this option.

Related commands

*Configure Scroll

Related SWIs

None

Related vectors

None

*Configure Quiet

Sets the beep at half volume

Syntax	`*Configure Quiet`
Parameters	None
Use	This command sets the configured volume for the bell to half its loudest volume. The change takes effect on the next reset.
Related commands	*Configure Loud
Related SWIs	OS_Byte 212 (SWI &06)
Related vectors	None

*Configure ScreenSize

Reserves an area of memory for screen display

Syntax

```
*Configure ScreenSize <n>|<mK>
```

Parameters

`<n>`	number of pages of memory; n<=127
`<mK>`	kilobytes of memory reserved

Use

This command reserves an area of memory for screen display. The default value is 80Kbytes on machines with 0.5Mbytes and 160Kbytes for machines with more than that.

Refer to OS_ChangeDynamicArea for information on how to change the screen memory allocation dynamically.

Note that the screen memory allocation should not be configured any greater than 480K, due to limitations in MEMC1 and MEMC1a.

Example

```
*Configure ScreenSize 160K
```

Related commands

None

Related SWIs

None

Related vectors

None

*Configure Scroll

Sets the screen to scroll upwards at the end of a line

Syntax

```
*Configure Scroll
```

Parameters

None

Use

This generates a newline whenever a character is printed at the end of a line. This is the default.

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.

Refer to VDU 23,16 for a lengthier description of scroll and to show how to dynamically set this option.

Related commands

*Configure NoScroll

Related SWIs

None

Related vectors

None

*Configure Sync

Selects the type of synchronisation for vertical sync output

Syntax

```
*Configure Sync <0|1>
```

Parameters

0 or 1	vertical or composite sync

Use

This selects vertical sync (0 parameter) or composite sync (1 parameter) on the vertical sync output of the video connector. For any monitor currently supplied for use with Acorn computers, the default should not be changed from 1.

Example

```
*Configure Sync 1
```

Related commands

None

Related SWIs

None

Related vectors

None

*Configure TV

Adjusts screen alignment and screen interlace

Syntax	`*Configure TV [<vert align> [[,] <interlace]]`
Parameters	`<vert align>` adjusts the vertical screen alignment 0 to 3 lines up, or 255 to 252 (1 to 4 lines down) `<interlace>` switches screen interlace on with 0, or off with 1
Use	This command sets the vertical alignment and interlace options. The default values are 0,1 (no vertical alignment offset and interlace off).
Example	`*Configure TV 0,1` the default value
Related commands	*TV
Related SWIs	None
Related vectors	None

*Pointer

Turns the mouse pointer on or off

Syntax	`*Pointer [0|1]`
Parameters	0 or 1 or nothing
Use	This command is used to switch on or off the pointer that appears on screen to reflect the mouse position. It can also be moved with OS_Word 21,5 if the mouse and pointer are unlinked. OS_Word 21,6 can read the pointer position at any time. No parameter or 1 will enable the pointer display, while 0 will disable it.
Example	`*Pointer 0` turn off the pointer
Related commands	None
Related SWIs	OS_Word 21 (SWI &07)
Related vectors	None

*ScreenLoad

Copies a sprite file into the graphics window

Syntax

```
*ScreenLoad <pathname>
```

Parameters

`<pathname>`	a valid pathname, specifying a sprite file

Use

This command copies the contents of a sprite file (for example, saved using *ScreenSave) into the graphics window, which is typically the whole screen.

Example

```
*ScreenLoad My.Pic
```

Related commands

*ScreenSave

Related SWIs

None

Related vectors

None

*ScreenSave

Copies graphics window to a file

Syntax	`*ScreenSave <pathname>`
Parameters	`<pathname>` a valid pathname, specifying a file
Use	This command copies the contents of the graphics window (typically the whole screen) and its palette to a file, which is saved as a sprite. This file can then be used by Paint or Draw.
Example	`*ScreenSave My.Pic`
Related commands	*ScreenLoad
Related SWIs	None
Related vectors	None

*Shadow

Causes the alternate bank of screen memory to be used

Syntax

```
*Shadow [0|1]
```

Parameters

0 or 1 or nothing

Use

*Shadow makes subsequent changes to the screen mode using the alternate bank of memory, called the shadow memory.

Passing 0 or no parameter causes this command to use the shadow bank on next mode change. Passing 1 causes the non-shadow bank to be used.

For the shadow bank to be used, there must be at least double the memory for the selected screen mode available in the screen area of memory.

Example

```
*Shadow 1
```

Related commands

*Configure ScreenSize

Related SWIs

None

Related vectors

None

*TV

Adjusts screen alignment and screen interlace

Syntax

```
*TV [<vert align> [[,] <interlace]]
```

Parameters

`<vert align>`	adjusts the vertical screen alignment 0 to 3 lines up, or 255 to 252 (1 to 4 lines down)
`<interlace>`	switches screen interlace on with 0, or off with 1

Use

This command sets the vertical alignment and interlace options. The default values are 0,1 (no vertical alignment offset and interlace off). These are set by *Configure TV.

Example

```
*TV 0,1
```

Related commands

*Configure TV

Related SWIs

None

Related vectors

None

Application Notes

Examples of ECF pattern use

In BBC/Master compatible mode

This section gives some examples of how you might set ECF patterns using the VDU 23,2-5... commands.

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 (left)	Green	2	0010
Pixel 2 colour (right)	White	7	0111

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

Resulting value = &1D (29)

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

Pixel 1 colour (left)	Yellow	2	10
Pixel 2 colour	Red	1	01
Pixel 3 colour	White	3	11
Pixel 4 colour (right)	Yellow	2	10

Bit	7	6	5	4	3	2	1	0
Pixel 1	1				0			
Pixel 2		0				1		
Pixel 3			1				1	
Pixel 4				1				0
Result	1	0	1	1	0	1	1	0

Resulting value = &B6 (182)

In RISC OS native mode

In RISC OS mode, 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 (left)	Green	2	0010
Pixel 2 colour (right)	White	7	0111

Bit	7	6	5	4	3	2	1	0
Right pixel	0	1	1	1				
Left pixel					0	0	1	0
Result	0	1	1	1	0	0	1	0

Resulting value = &72 (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 (left)	Yellow	2	10
Pixel 2 colour	Red	1	01
Pixel 3 colour	White	3	11
Pixel 4 colour (right)	Yellow	2	10

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 (182)

Further examples of ECF patterns

Here are examples of how to produce a pattern of alternating red (colour 1) lines and white (colour 7) lines (with the default palette). Each of the VDU 23,2 or VDU 23,12 commands alters ECF pattern 1 to cause the same effect.

in a 2 colour mode (black and white only available):

```
VDU 23,12,1,1,0,0,1,1,0,0
VDU 23,2,&FF,0,&FF,0,&FF,0,&FF,0
VDU 23,17,4,1| has no effect
```

in a 4 colour mode:

```
VDU 23,12,1,1,3,3,1,1,3,3
VDU 23,2,&0F,&FF,&0F,&FF,&0F,&FF,&0F,&FF
after VDU 23,17,4,1|
VDU 23,2,&55,&FF,&55,&FF,&55,&FF,&55,&FF
```

in a 16 colour mode:

```
VDU 23,12,1,1,7,7,1,1,7,7
VDU 23,2,3,&3F,3,&3F,3,&3F,3,&3F
after VDU 23,17,4,1|
VDU 23,2,&11,&77,&11,&77,&11,&77,&11,&77
```

in a 256 colour mode:

```
VDU 23,12,&C3,&FF,&C3,&FF,&C3,&FF,&C3,&FF
VDU 23,2,&17,&FF,&17,&FF,&17,&FF,&17,&FF
VDU 23,17,4,1| has no effect
```

Sprites

Introduction

A sprite is an area of memory that can be treated like a small block of screen memory. It contains a graphic shape made up of an array of pixels.

A sprite has the following attributes:

- a name used to identify the sprite, up to 12 characters in length
- the number of the screen mode whose format the sprite imitates
- a height and a width
- optionally, a transparency mask.
- optionally, a palette defining the colours used in the sprite

If the sprite has a transparency mask, you can cause certain pixels in the sprite not to be written to the existing screen display. By using this mask, you can effectively make a sprite any shape.

A sprite can be defined by grabbing some or all of the screen, or defining it a pixel at a time or by making the VDU plot operations go into a sprite instead of the screen memory.

Once defined, a sprite can be manipulated in many ways, such as having rows and columns inserted or deleted, flipping it about the x or y axis and changing the colour of particular pixels.

A sprite can be plotted onto the screen scaled to any size, and its colours can be altered using a lookup table.

Sprites are stored in sprite files, which may contain one or more sprites with different names.

Overview

Sprite memory areas

RISC OS can use sprites from the *system sprite area*, or from any number of *user sprite areas*.

System sprite area

The first is the system sprite area, which is defined by the kernel. Its size can be controlled by a slider in the task manager application on the desktop.

This area is public and can be accessed from any program or module, so is a convenient place to experiment using sprites. However, you should not use the system sprite area in commercial applications and should instead use a combination of the Wimp's common sprite pool and user sprite area as appropriate.

Note that the Sprite module * Commands only work with sprites in the system sprite area.

User sprite area

Alternatively, an application or a module may reserve its own space. This is private space, which can only be used by the application or module that reserved it. For example, the Wimp has a shared sprite pool, which is passed to OS_SpriteOp as a user area.

Unlike the system area, there can be several user areas which are referenced via pointers to the start of the areas. In user areas, as well as being able to refer to a sprite by name, you can also refer to it by address. This plainly will be much faster, since there is no overhead to search through the available names.

Memory operations

With the sprite module, it is possible to issue calls to:

- clear a sprite area
- check how large an area is and how many sprites are in it
- scan through the list of names of sprites in an area

File operations

Sprites can be loaded and saved to any valid pathname. The simplest way of doing this is to use the calls to load or save the current graphics window as a single sprite file.

For more sophisticated control, a sprite area (system or user) can be saved, or loaded. It is also possible to merge a sprite file with what is already in memory.

Sprite files can be edited by the Paint application.

Creating sprites

You can create a blank sprite of a specified height and width. Subsequently, individual pixels can be changed within it.

You also have various ways of grabbing some or all of the graphics window and putting it into a sprite.

The various sprite editing utilities all use one or other of these techniques.

Mask control

The mask can be enabled and disabled as required. Like a sprite, it can have individual pixels set or cleared. A sprite may have up to 256 colours (64 palette entries stored), depending on which mode it was created in; the mask pixels are either on (solid), in which case the pixel colour is used, or off (transparent), in which case it is not plotted.

VDU output to sprite

The other way of writing to a sprite or its mask is to redirect the VDU operations to a sprite. This means that the sprite rectangle is treated like a graphics window, putting data into the sprite in the same format as the screen memory.

Sprite manipulation

Once a sprite is in memory, it can be manipulated in a number of ways, for example you can:

- rename, copy, delete the sprite or append it to another sprite
- insert or delete rows and columns
- flip about the x or y axis
- change an individual pixel's colour.

Plotting a sprite

There are several ways of plotting a sprite into the screen memory. There is a SWI that will simply plot the sprite. You can also plot it using the mask if one is attached to it. The scale of the sprite can be changed to be any desired size. Thus, zooming into a sprite is made very easy.

The anti-aliasing technique used by the font manager with characters can be used here with sprites. A range of close colours are used to shade the sprite, which can be plotted with or without a mask, and scaled to any size.

Technical Details

Common parameters

Several kinds of parameters are used by many SWIs within the sprite module. Rather than repeating their definitions each time, they are described here.

Pointer to control block of sprite area and sprite pointer

Many of the sprite SWIs use a pointer to control block of sprite area parameter in R1 or that and a sprite pointer in R2. When either of these appear, then bits 8 and 9 in R0 control how these two registers are interpreted.

R0 bit 8 & 9 values	R1 effect	R2 effect
00 (+0)	not used (system sprite area used)	pointer to sprite name
01 (+256)	pointer to user sprite area	pointer to sprite name
10 (+512)	pointer to user sprite area	pointer to sprite
11 (+768) is invalid		

Note that the sprite names are null terminated.

For example OS_SpriteOp 256+33,CBlock,NamePtr will interpret CBlock as a pointer to the user sprite area and use NamePtr as a pointer to the name of the sprite to use within that area.

Using a pointer to a sprite in the user area (R0+512) is the quickest way of using sprites, because the string lookup doesn't need to be done.

Scale factors

The scale factor will change the size of a sprite. It is a pointer to a block of four words with the following elements:

Offset	Meaning
0	x multiplier
4	y multiplier
8	x divisor
12	y divisor

The size of the specified sprite on the screen when it has been plotted in pixels (NOT OS units), is multiplied by the magnitude and divided by the divisor. ie.:

x pixel size = x start size (in pixels) * x magnitude / x divisor
y pixel size = y start size (in pixels) * y magnitude / y divisor

If the plot action is using an ECF pattern, then the pattern will not be scaled up with the sprite. This is so that the patterning will be correct when used with a large scale factor. See the chapter entitled *VDU drivers* for a description of ECF patterns.

If the pointer is zero, then no scaling is performed. ie. 1:1 scale.

Pixel translation table

This allows a logical colour to be substituted for each colour in the sprite. It is a pointer to a table of bytes. The number of bytes in the table depends on the number of colours in the mode in which the sprite was created.

A pixel of colour N in the sprite will be translated to the Nth entry in the pixel translation table. The first entry in the table is at offset 0 (ie the 0th colour). So Colour 3 in a pixel will get the value 3 bytes into the table and use that as its logical colour.

If the pointer is zero, then the colours in the sprite will be used. However, if the destination bits per pixel is less than the source bits per pixel, you will get an error.

The wimp uses a similar system to provide mode independence. See Wimp_ReadColourTable in the chapter entitled *Window Manager* for details.

The ColourTrans module provides facilities for translation table calculations. For more information refer to the chapter entitled *ColourTrans*.

Plot action

The plot action is the way in which pixels are plotted onto the screen. Some SWIs use the VDU 18 setting, and others can be passed the number directly. In either case, the format is the same, apart from bit 3 (&08)

Value	Action
0	Overwrite colour on screen
1	OR with colour on screen
2	AND with colour on screen
3	exclusive OR with colour on screen
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND with colour on screen with NOT of sprite pixel colour
7	OR with colour on screen with NOT of sprite pixel colour

&08	If set, then use the mask, otherwise don't
&10	ECF pattern 1
&20	ECF pattern 2
&30	ECF pattern 3
&40	ECF pattern 4
&50	Giant ECF pattern (patterns 1 - 4 placed side by side)

Save area

When output is switched to a sprite or its mask, it is possible to save the VDU context in a save area. The save area passed is where the state that has just been entered will be saved if *another* redirection of VDU output is made.

The save area is a block of memory, the size of which is obtained from OS_SpriteOp 62. The contents cannot be directly manipulated, but this is a list of the things that it stores:

- ECF patterns, BBC/Native ECF flag, ECF origin
- Dotted line pattern and length, and current position in pattern
- Graphics foreground and background actions, colours and tints
- Text foreground and background colours and tints
- Graphics and text window definitions
- Graphics origin
- Graphics cursor and two previous positions
- Text and input cursor positions
- VDU status (VDU 2 state, page mode, windowing, shadowing, VDU 5 mode, cursor editing state and VDU disabled/enabled)
- VDU queue and queue pointer
- Character sizes and spacings
- Changed box coordinates and status
- Wrch destinations flag
- Spool handle

Mode variables are reconstituted from the sprite mode number or the display mode number as appropriate.

The kernel maintains a save area for the screen (ie the system save area with a value 1). Therefore, if you swap output to a sprite, perform some operations and swap back, it will not be necessary to allocate a save area.

A save area that has not yet been used must have a zero in the first word. Once it has been used, then this is set to a non-zero value, so that when it is next passed to OS_SpriteOp 60 or OS_SpriteOp 61 the graphics state will be restored from it, rather than being set to the default state.

The use of save areas allows the VDU 'context' to be switched between various destinations, so that each area has its own separate VDU state.

Here are a couple of examples highlighting the above points. The first example shows how to set-up a once-off drawing into a sprite:

```
SYS "OS_SpriteOp",256+60,myarea,mysprite$,0 TO r0,r1,r2,r3
REM we don't need a save area, because nobody can swap output away from
REM our sprite; and we won't want to restore the state we're in when
REM we've finished our work on the sprite.
.... do whatever graphics we want ....
SYS "OS_SpriteOp",r0,r1,r2,r3
REM whatever output state was in force on entry is now restored
```

The second example shows how to draw into a sprite, interact with the user, while maintaining ECF patterns etc:

```
SYS "OS_SpriteOp",256+62,myarea,mysprite$ TO ,,,size
DIM sarea size
sarea!0=0 : REM mark as unset
REPEAT
  SYS"OS_SpriteOp",256+60,myarea,mysprite$,sarea TO r0,r1,r2,r3
    .... work on the sprite
  SYS "OS_SpriteOp",r0,r1,r2,r3: REM return to previous output
  REM at this point, our save area has been filled with our state;
  REM the next time we switch output to our sprite the OS variables
  REM will therefore be reset from it.
    ... talk to the user ....
UNTIL bored
```

Memory operations

To initialise the system sprite area, you can call OS_SpriteOp 9 or *SNew. To change the system sprite area size, you can call OS_ChangeDynamicArea (SWI &2A); you can also change the configured size of this area (which is used on a hard reset) by calling *Configure SpriteSize.

In order to setup a user sprite area, you must first allocate space for it using the usual memory allocation calls. You must then set up the header for the area before you call OS_SpriteOp 9 to initialise it as a sprite area.

Reading a sprite area

To check the state of a sprite area, *SInfo or OS_SpriteOp 8 will tell you how large the area is, how much has been used and how many sprites are in it. *SInfo will, of course, only work with the system area.

Finding the names of sprites

*SList will list the names of all sprites in the system area. OS_SpriteOp 13 allows you to find the name of a sprite given its number in the list. You would call OS_SpriteOp 8 first to find out how many sprites there are and then use this call to get the names one at a time.

File operations

The simplest sprite file operations are screen save and load. The screen save will take the entire graphics window and convert it into a sprite file. *ScreenSave and OS_SpriteOp 2 will perform this operation. *ScreenLoad and OS_SpriteOp 3 will load it back again, aligned with the bottom left hand corner of the current graphics window.

There is also a set of operations based around loading and saving sprite areas to a file. *SLoad and OS_SpriteOp 10 will load a sprite file into an initialised sprite area and set up all the pointers within it. To save, *SSave and OS_SpriteOp 12 will create a sprite file and write all the sprites from the specified sprite area into it.

The sprite load operations will delete all sprites currently in memory. If you wish to keep them, then *SMerge and OS_SpriteOp 11 will merge the sprite file sprites with those in memory. Any name clashes will result in the file sprite replacing the memory one.

Creating a sprite

There are two main ways of creating a sprite. You can grab a piece of screen memory using OS_SpriteOp 14 or 16, or *SGet. Alternatively, you can create a blank sprite with OS_SpriteOp 15 to be subsequently filled in. With this blank sprite, you can alter individual pixels or you can direct VDU operations into it. These are discussed later.

Creating a mask

To create a mask, OS_SpriteOp 29 must be used. It will initialise all the pixels solid, so that all of the sprite is plotted. You must alter it afterwards to set the mask that you require.

Sprite manipulation

The contents of a sprite may be manipulated in many ways.

Copy, rename or delete

You can copy, rename or delete a sprite in the following ways:

- To make a copy of a sprite, OS_SpriteOp 27 or *SCopy can be used. They will return an error if the designated name already exists.
- To rename a sprite, OS_SpriteOp 26 or *SRename can be used. Again, the same error condition applies to existing destination names.
- To delete a sprite, its mask and palette, OS_SpriteOp 25 or *SDelete can be used You can delete the mask of a sprite only, by calling OS_SpriteOp 30. Free space is automatically reclaimed in the sprite area.

Insert and delete row or column

You can insert and delete rows and columns at any place you wish in the sprite. These are the operations that you need to do this:

- OS_SpriteOp 31 to insert a row
- OS_SpriteOp 32 to delete a row
- OS_SpriteOp 45 to insert a column
- OS_SpriteOp 46 to delete a column

Axis flipping

A sprite can be flipped about its x or y axis. Flipped about the x axis using OS_SpriteOp 33 or *SFlipX will make it appear upside down. Flipping about the y axis with OS_SpriteOp 47 or *SFlipY will make it look back to front.

Remove wastage

If a sprite is not a whole number of words wide, it is possible that part of each row on the left and right is "wasted"; that is, it does not form part of the sprite image. To remove this wastage, OS_SpriteOp 54 will align the sprite with the left hand side. If more than 32 free bits are on the right of the sprite, then these words will be removed.

Appending

Sprites can be tacked together, either horizontally or vertically using OS_SpriteOp 35. No extra memory is used to do this.

Reading and altering pixels

To check the size of a sprite, OS_SpriteOp 40 will return its width, height, screen mode and whether it has a mask or not.

If you wish to read a pixel in a sprite, then OS_SpriteOp 41 will return colour and tint for a given x and y coordinate in the sprite. To write a pixel colour, OS_SpriteOp 42 must be used. It is given the coordinates, colour and tint to use.

Reading and altering the mask

Similar to these last two SWIs, OS_SpriteOp 43 will read a mask bit and OS_SpriteOp 44 will write it. Remember that a mask has the same number of bits per pixel as the image, but that the bits for each pixel must either be all set, or all clear.

VDU output to sprites

The VDU drivers can be directed to put their output into a sprite instead of the screen. OS_SpriteOp 60 will switch output to a sprite or to the screen. OS_SpriteOp 61 will switch output to a mask or the screen.

The save area described earlier is used by these calls. The space required for a save area can be determined by calling OS_SpriteOp 62.

Plotting sprites

To plot a sprite on the screen, OS_SpriteOp 28 and 34 are the simplest to use. They plot the sprite at the current graphics cursor position, using the current GCOL action. OS_SpriteOp 48 and 49 are similar, but the coordinates and GCOL action are instead passed explicitly.

Scaled plotting

A sprite can be plotted at any magnification using OS_SpriteOp 50 and 52.

Like these SWIs, OS_SpriteOp 53 will plot a sprite using scale factors and a translation table, but it uses the anti-aliased colour technique that the font manager uses for characters.

OS_SpriteOp 51 will paint a character onto the screen using scale factors.

Format of a sprite area

The format of a sprite area is as follows:

Control Block	Extension Area (Optional)	Sprite	Sprite	Free Space

The sprite area control block contains the following:

Word	Contents
1	Byte offset to last byte+1 (ie total size of sprite area)
2	Number of sprites in area
3	Byte offset to first sprite
4	Byte offset to first free word (ie byte after last sprite)
5 ...	Extension words (usually null)

The above offsets are relative to the start of the sprite area control block.

The format of the file created by a *ScreenSave or *SSave command is the same as a sprite area but without word 1 of the control block. This is because it is only valid in memory.

Format of a sprite

The format of a sprite is as follows:

Control Block	Palette Area (Optional)	Sprite Image	Plotting Mask (Optional)

The Sprite Control Block contains the following:

Word	Content
1	Offset to next sprite
2 - 4	Sprite name, up to 12 characters with trailing zeroes
5	Width in words –1
6	Height in scan lines –1
7	First bit used (left end of row)
8	Last bit used (right end of row)
9	Offset to sprite image
10	Offset to transparency mask or offset to sprite image if no

	mask
11	Mode sprite was defined in
12 ...	Palette data (optional)

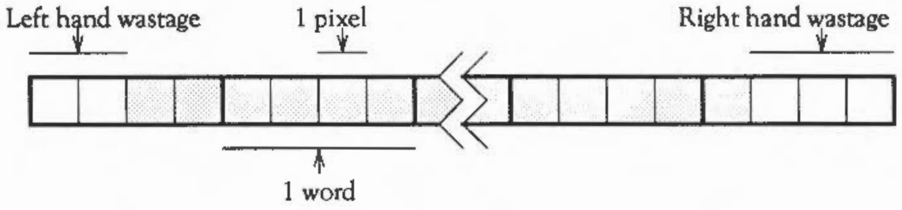
The size of the palette data block depends on the number of bits per pixel in the sprite's mode, since there will be one entry for each potential logical colour. 256 colour modes are the exception to this rule, because there are only 16 palette registers.

Note that 256 colour sprites created by *ScreenSave actually have 64 palette entries; the last 16 are the ones that are actually enforced.

Each entry is two words long. These are the words returned from OS_ReadPalette (SWI &2F). The format of these words is described with this SWI in the chapter entitled *VDU drivers*.

Format of a sprite image

The format of a sprite image is as follows:



The image contains the rows of the sprite from top to bottom, all word-aligned. Each pixel is a group of <bytes per character> bits (see the section on VDU variables in the chapter entitled *VDU drivers*). The least significant pixel in a word is the left-most one on the screen.

Note that in the diagram above, bit 0 of each word has been shown on the *left*, and bit 31 has been shown on the *right*; this is to clarify how wastage occurs. Note also that there will not necessarily be 4 pixels per word.

Format of a sprite mask

A sprite mask is the same size as the corresponding sprite image, and the same bits refer to each pixel. In the mask, the bits of each pixel must either all be set (the sprite's pixel is solid) or all be cleared (the pixel is transparent)

VDU commands

There are ways of selecting a sprite so that it can subsequently be used by the VDU commands described below to plot sprites.

The VDU commands are included for compatibility only and in RISC OS are of very little use since they only allow access to the system sprite area, whereas you will more likely be using user sprite areas.

Any programs being written for the Wimp must not use these VDU commands because there is only one location storing the setting for the selected sprite, not one per process.

As well as *SChoose and OS_SpriteOp 24, a sprite can be selected for VDU use by:

```
VDU 23,27,m,n|
```

where:	m = 0	is equivalent to *SChoose n.
	m = 1	is equivalent to *SGet n

Plotting a sprite

Once a sprite has been selected by either of the three techniques above, it can be plotted using:

```
VDU 25,232 - 239,x;y;
```

The range of eight plot numbers are the standard plot options as defined in VDU 25 in the chapter entitled *VDU drivers*. x and y are in OS coordinates.

OS_SpriteOp (SWI &2E)

Controls the sprite system

On entry

R0 = reason code
Other registers depend on reason code

On exit

R0 preserved
Other registers depend on reason code

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call controls the sprite system. It is indirected through SpriteV.

The particular action of OS_SpriteOp is given by the reason code in R0 as follows:

R0	Action
2	Screen save
3	Screen load
8	Read area control block
9	Initialise sprite area
10	Load sprite file
11	Merge sprite file
12	Save sprite file
13	Return name
14	Get sprite
15	Create sprite
16	Get sprite from user co-ordinates
24	Select sprite
25	Delete sprite
26	Rename sprite
27	Copy sprite
28	Put sprite

29	Create mask
30	Remove mask
31	Insert row
32	Delete row
33	Flip about x axis
34	Put sprite at user coordinates
35*	Append sprite
36*	See pointer shape
40	Read sprite information
41	Read pixel colour
42	Write pixel colour
43	Read pixel mask
44	Write pixel mask
45	Insert column
46	Delete column
47	Flip about y axis
48	Plot sprite mask
49	Plot mask at user coordinates
50	Plot mask scaled
51*	Paint character scaled
52*	Put sprite scaled
53*	Put sprite grey scaled
54	Remove lefthand wastage
60	Switch output to sprite
61	Switch output to mask
62	Read save area size

For details of each of these reason codes, see below.

Note that the reason codes marked with an asterisk cannot be used with RISC OS 2.00 version of the SpriteExtend module.

Related SWIs

None

Related vectors

SpriteOpV

OS_SpriteOp 2 (SWI &2E)

Screen save

On entry

R0 = 2
R2 = pointer to pathname
R3 = palette flag (0 not to save, 1 to save)

On exit

R0 preserved
R2 preserved
R3 preserved

Use

This saves the current graphics window as a sprite file. The file contains a single sprite called 'screendump'. If R3 is 0, no palette information is saved with the file; if it is 1, the current palette is saved. It is equivalent to *ScreenSave.

See reason code 3 to reverse the operation and load a screen.

Related SWIs

OS_SpriteOp 3 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 3 (SWI &2E)

Screen load

On entry

R0 = 3
R2 = pointer to pathname

On exit

R0 preserved
R2 preserved

Use

This plots a sprite directly from a file to the screen. It changes mode if necessary and sets the palette to the setting held in the file. The sprite is plotted at the bottom left of the graphics window. After a mode change, this is the bottom left-hand corner of the screen. It is equivalent to *ScreenLoad.

See reason code 2 to reverse the operation and save a screen.

Related SWIs

OS_SpriteOp 2 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 8 (SWI &2E)

Read area control block

On entry

R0 = 8
R1 = pointer to control block of sprite area

On exit

R0 preserved
R1 preserved
R2 = total size of sprite area in bytes
R3 = number of sprites in area
R4 = byte offset to the first sprite
R5 = byte offset to the first free word

Use

This returns all the information contained in the control block of a sprite area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 9 (SWI &2E)

Initialise sprite area

On entry

R0 = 9
R1 = pointer to control block of sprite area

On exit

R0 preserved
R1 preserved

Use

This initialises a sprite area. It is equivalent to *SNew when used with the system area.

If you are initialising a user sprite area, then you must first initialise two words in the area header:

Address	Contents of word
area + 0	total size of area
area + 8	offset to first sprite (= 16, if the extension area is null)

Setting bit 8 or 9 of R0 alters the interpretation of R1 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 10 (SWI &2E)

Load sprite file

On entry

R0 = 10 (&0A)
R1 = pointer to control block of sprite area
R2 = pointer to pathname

On exit

R0 preserved
R1 preserved
R2 preserved

Use

This loads the sprite definitions contained in the file into the sprite area, overwriting any definitions stored there already. It is equivalent to *SLoad when used with the system area.

The first word of the sprite area must be initialised to its size before you call this SWI.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 11 (SWI &2E)

Merge sprite file

On entry

R0 = 11 (&0B)
R1 = pointer to control block of sprite area
R2 = pointer to pathname

On exit

R0 preserved
R1 preserved
R2 preserved

Use

This merges the sprite definitions contained in the file with those in the sprite area. It is equivalent to *SMerge when used with the system area.

Note that there must be enough free space in the sprite area to hold both the new file and the original sprites, since it is only after the new file has been loaded that any of the original sprites are replaced by new ones that have the same name.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 12 (SWI &2E)

Save sprite file

On entry

R0 = 12 (&0C)
R1 = pointer to control block of sprite area
R2 = pointer to pathname

On exit

R0 preserved
R1 preserved
R2 preserved

Use

This saves the contents of a sprite area to a file. It is equivalent to *SSave when used with the system area.

The first word of the sprite area (its size) is not saved.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 13 (SWI &2E)

Return name

On entry

R0 = 13 (&0D)
R1 = pointer to control block of sprite area
R2 = pointer to buffer
R3 = maximum name length (ie. buffer size)
R4 = sprite number (position in workspace – the first one is numbered 1)

On exit

R0 preserved
R1 preserved
R2 preserved
R3 = name length
R4 preserved

Use

This returns the name of the sprite whose position in the workspace (eg 3 for the third sprite) is given in R4. The name is placed in the buffer pointed to by R2 as a null-terminated string, the length of which is returned in R3.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 14 (SWI &2E)

Get sprite

On entry

R0 = 14 (&0E)
R1 = pointer to control block of sprite area
R2 = pointer to sprite name
R3 = palette flag (0 to exclude palette data, 1 to include it)

On exit

R0 preserved
R1 preserved
R2 = address of sprite (if in user sprite area)
R3 preserved

Use

This defines the sprite identified to be the current contents of an area of the screen. It is delimited by the current and old cursor positions (inclusive). If the sprite already exists, it is overwritten. It is equivalent to *SGet when used with the system area.

Any part of the designated area which lies outside the current graphics window is filled with the current background colour in the sprite.

Setting bit 8 of R0 alters the interpretation of R1 – see the start of the earlier section on *Technical Details* for a description. You must not call this SWI with bit 9 of R0 set; that is, R2 must always point to a sprite name.

Related SWIs

OS_SpriteOp 16 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 15 (SWI &2E)

Create sprite

On entry

R0 = 15 (&0F)
R1 = pointer to control block of sprite area
R2 = pointer to sprite name
R3 = palette flag (0 to exclude palette data, 1 to include it)
R4 = width in pixels
R5 = height in pixels
R6 = mode number

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved
R5 preserved
R6 preserved

Use

This creates a blank sprite of a given size.

Setting bit 8 of R0 alters the interpretation of R1 – see the start of the earlier section on *Technical Details* for a description. You must not call this SWI with bit 9 of R0 set; that is, R2 must always point to a sprite name.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 16 (SWI &2E)

Get sprite from user coordinates

On entry

R0 = 16 (&10)
R1 = pointer to control block of sprite area
R2 = pointer to sprite name
R3 = palette flag (0 to exclude palette data, 1 to include it)
R4 = left hand edge OS screen coordinate (inclusive)
R5 = bottom edge OS screen coordinate (inclusive)
R6 = right hand edge OS screen coordinate (inclusive)
R7 = top edge OS screen coordinate (inclusive)

On exit

R0 preserved
R1 preserved
R2 = address of sprite (if in user sprite area)
R3 preserved
R4 preserved
R5 preserved
R6 preserved
R7 preserved

Use

This picks up an area of the screen, which is delimited by the coordinates supplied (inclusive), as a sprite. If the sprite already exists, it is overwritten.

Any part of the designated area which lies outside the current graphics window is filled with the current background colour in the sprite.

Setting bit 8 of R0 alters the interpretation of R1 – see the start of the earlier section on *Technical Details* for a description. You must not call this SWI with bit 9 of R0 set; that is, R2 must always point to a sprite name.

Related SWIs

OS_SpriteOp 14 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 24 (SWI &2E)

Select sprite

On entry

R0 = 24 (&18)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 preserved
R1 preserved
R2 = address of sprite (if in user sprite area), otherwise preserved

Use

Select a particular sprite for subsequent plotting. That is, the VDU 25,232-239 commands will use the selected sprite. It is equivalent to *SChoose when used with the system area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 25 (SWI &2E)

Delete sprite

On entry

R0 = 25 (&19)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 preserved
R1 preserved
R2 preserved

Use

This deletes the definition of a particular sprite. It is equivalent to *SDelete when used with the system area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 26 (SWI &2E)

Rename sprite

On entry

R0 = 26 (&1A)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = pointer to new name

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved

Use

This changes the name of a sprite. An error is produced if a sprite of the new name already exists in the same sprite area. It is equivalent to *SRename when used with the system area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 27 (SWI &2E)

Copy sprite

On entry

R0 = 27 (&1B)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = pointer to new name

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved

Use

This copies a sprite within a sprite area. An error is produced if a sprite of the new name already exists in the same sprite area. It is equivalent to *SCopy when used with the system area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 28 (SWI &2E)

Put sprite

On entry

R0 = 28 (&1C)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R5 = plot action

On exit

R0 preserved
R1 preserved
R2 preserved
R5 preserved

Use

This plots the sprite identified with its bottom left corner at the current graphics cursor position using the plot action specified in R5:

Value	Action
0	Overwrite colour on screen
1	OR with colour on screen
2	AND with colour on screen
3	exclusive OR with colour on screen
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND with colour on screen with NOT of sprite pixel colour
7	OR with colour on screen with NOT of sprite pixel colour
&08	If set, then use the mask, otherwise don't
&10	ECF pattern 1
&20	ECF pattern 2
&30	ECF pattern 3
&40	ECF pattern 4
&50	Giant ECF pattern (patterns 1 - 4 placed side by side)

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs	OS_SpriteOp 48 (SWI &2E)
Related vectors	SpriteV

OS_SpriteOp 29 (SWI &2E)

Create mask

On entry

R0 = 29 (&1D)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 preserved
R1 preserved
R2 preserved

Use

This creates a mask for the specified sprite with all pixels set to be solid.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 30 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 30 (SWI &2E)

Remove mask

On entry

R0 = 30 (&1E)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 preserved
R1 preserved
R2 preserved

Use

This removes the mask definition for a given sprite.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 29 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 31 (SWI &2E)

Insert row

On entry

R0 = 31 (&1F)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = row number

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved

Use

This inserts a row in the sprite at the position identified, shifting all rows above it up one. All pixels in the new row are set to colour zero. Rows are numbered from the bottom upwards with the bottom row being number zero. If the row number is equal to the height of the sprite it will go on top. Any value above this will generate an error.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 32, 45 and 46 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 32 (SWI &2E)

Delete row

On entry

R0 = 32 (&20)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = row number

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved

Use

This deletes a row in the sprite at the position identified, shifting all rows above it down one. Rows are numbered from the bottom upwards with the bottom row being number zero. If the row number is greater than or equal to the height of the sprite it will generate an error.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 31, 45 and 46

Related vectors

SpriteV

OS_SpriteOp 33 (SWI &2E)

Flip about x axis

On entry

R0 = 33 (&21)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 preserved
R1 preserved
R2 preserved

Use

This takes the sprite identified and reflects it about the x axis so that it is upside down. Thus, its top row on entry becomes the bottom row on exit, and so on.

It is equivalent to *SFlipX when used on the system area sprites.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 47 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 34 (SWI &2E)

Put sprite at user coordinates

On entry

R0 = 34 (&22)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate
R4 = y coordinate
R5 = plot action

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved
R5 preserved

Use

This plots a sprite at the external coordinates supplied, using the plot action supplied in R5:

Value	**Action**
0	Overwrite colour on screen
1	OR with colour on screen
2	AND with colour on screen
3	exclusive OR with colour on screen
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND with colour on screen with NOT of sprite pixel colour
7	OR with colour on screen with NOT of sprite pixel colour
&08	If set, then use the mask, otherwise don't
&10	ECF pattern 1
&20	ECF pattern 2
&30	ECF pattern 3
&40	ECF pattern 4
&50	Giant ECF pattern (patterns 1 - 4 placed side by side)

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 35 (SWI &2E)

Append sprite

On entry

R0 = 35 (&23)
R1 = pointer to control block of sprite area
R2 = sprite pointer 1
R3 = sprite pointer 2
R4 = 0 to merge horizontally, or 1 to merge vertically

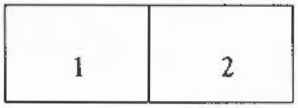
On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved

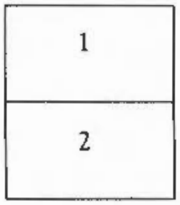
Use

This call can be used to merge two sprites of the same height or width into one sprite, tacking them together vertically or horizontally.

The sprites are appended horizontally in the following order:



The sprites are appended vertically in the following order:



The result of the merge is stored in sprite 1 and sprite 2 is deleted. Thus the merge does not consume any extra memory.

Attempting to merge two sprites with different vertical or horizontal sizes will result in an error.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 36 (SWI &2E)

Set pointer shape

On entry

R0 = 36 (&24)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = bitfield (see below)
R4 = x offset of active point
R5 = y offset of active point
R6 = scale factors (0 to scale for the mode)
R7 = pixel translation table

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved

Use

This call sets any of the hardware pointer shapes to be programmed from a sprite, with some degree of mode independence. ie. the aspect ratio is catered for.

Note that in high resolution monochrome modes (eg. mode 23), the pointer shape resolution is four times worse horizontally than the pixel resolution, and only colours 0, 1 and 3 can be used in the pointer shape definition. This call will cater for this problem by halving the width of the pointer, so that it is still possible to see what it is, although the pointer will be twice as wide as usual.

R3 on entry is a bitfield composed of the following fields:

Bit	**Meaning**
0-3	pointer shape number, currently in the range 1 - 4
4	if clear, then set the pointer shape data
5	if clear, then set the palette from the sprite
6	if clear, then program the pointer shape number

Bits 4, 5, and 6 of this bitfield can be used to defer certain aspects of this call until later. For example, if you wanted to set up the pointer shape without displaying the pointer, bits 5 and 6 would be set.

The coordinates in R4 and R5 are relative pixels from the top left corner of the sprite.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_Word 8 (SWI &07), Wimp_SetPointerShape (SWI &400D8)

Related vectors

SpriteV

OS_SpriteOp 40 (SWI &2E)

Read sprite information

On entry

R0 = 40 (&28)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 preserved
R1 preserved
R2 preserved
R3 = width in pixels
R4 = height in pixels
R5 = mask status (0 for no mask, 1 for mask)
R6 = screen mode in which the sprite was defined

Use

This returns information about the sprite, giving its width and height in pixels, whether the sprite has a mask and the screen mode in which the sprite was defined.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 41 (SWI &2E)

Read pixel colour

On entry

R0 = 41 (&29)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate (in pixels)
R4 = y coordinate (in pixels)

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved
R5 = colour
R6 = tint

Use

Given x and y coordinates in R3 and R4 (in pixels relative to the bottom left of the sprite definition), this call returns the current colour of the pixel at that position.

The colour and tint returned depends on the mode. If it is not a 256 colour mode, then colour is from zero to the number of colours–1 and tint is zero. In 256 colour modes, the colour is from 0 to 63 and tint is either 0, 64, 128 or 192.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 42 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 42 (SWI &2E)

Write pixel colour

On entry

R0 = 42 (&2A)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate
R4 = y coordinate
R5 = colour
R6 = tint

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved
R5 preserved
R6 preserved

Use

Given x and y coordinates (in pixels from the bottom left of the sprite definition), and colour and tint in R5 and R6, this call sets the pixel at the position given to that colour.

The colour and tint values used depend on the mode. If it is not a 256 colour mode, then colour is from zero to the number of colours–1 and tint is ignored. In 256 colour modes, the colour is from 0 to 63 and tint is either 0, 64, 128 or 192. ie. only bits 6 and 7 are used.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 41 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 43 (SWI &2E)

Read pixel mask

On entry

R0 = 43 (&2B)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate
R4 = y coordinate

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved
R5 = mask status (0 = transparent, 1 = solid)

Use

Given x and y coordinates in R3 and R4 (in pixels relative to the bottom left of the sprite definition), this call returns the current state of the mask at that position.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 44 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 44 (SWI &2E)

Write pixel mask

On entry

R0 = 44 (&2C)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate
R4 = y coordinate
R5 = mask status (0 = transparent, 1 = solid)

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved
R5 preserved

Use

Given x and y coordinates (in pixels from the bottom left of the sprite definition), and mask state in R5, this call sets the pixel at the position given to that mask.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 43 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 45 (SWI &2E)

Insert column

On entry

R0 = 45 (&2D)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = column number

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved

Use

This inserts a column at the position identified, shifting all columns after it one place to the right. The new column is set to have either transparent or colour pixels, depending on whether the sprite has a mask or not. Columns are numbered from the left with the left-hand one being number zero.

If the column number is equal to the width of the sprite it will go after the right hand side. Any value above this will generate an error.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 31, 32 and 46 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 46 (SWI &2E)

Delete column

On entry

R0 = 46 (&2E)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = column number

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved

Use

This deletes a column from the position identified, shifting all columns after it one place to the left. Columns are numbered from the left with the left-hand one being number zero.

If the column number is greater than or equal to the width of the sprite it will generate an error.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 31, 32 and 45 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 47 (SWI &2E)

Flip about y axis

On entry

R0 = 47 (&2F)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 preserved
R1 preserved
R2 preserved

Use

This takes the sprite identified and reflects it about the y axis so that it is facing in the opposite direction. Thus, its leftmost column on entry becomes the rightmost column on exit, and so on.

It is equivalent to *SFlipY when used with the system sprite area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 33 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 48 (SWI &2E)

Plot sprite mask

On entry

R0 = 48 (&30)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 preserved
R1 preserved
R2 preserved

Use

This plots a sprite mask in the background colour and action with its bottom left corner at the graphics cursor position. That is, all 1 bits in the mask are plotted in the background colour and action, and all 0 bits are ignored. If the sprite has no mask, a solid rectangle the same size as the sprite is drawn in the current background colour and action (as if there was a mask which was completely solid).

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 28 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 49 (SWI &2E)

Plot mask at user coordinates

On entry

R0 = 49 (&31)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate
R4 = y coordinate

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved

Use

This plots in the background colour and action through a sprite mask at the external coordinates supplied.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 48 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 50 (SWI &2E)

Plot mask scaled

On entry

R0 = 50 (&32)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate to plot at
R4 = y coordinate to plot at
R6 = scale factors

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved
R6 preserved

Use

A sprite mask is plotted on the screen, using the current background colour and action and the scaling factors provided.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 51 (SWI &2E)

Paint character scaled

On entry

R0 = 51 (&33)
R1 = character code
R3 = x coordinate to plot
R4 = y coordinate to plot
R6 = scale factors

On exit

R0 preserved
R1 preserved
R3 preserved
R4 preserved
R6 preserved

Use

The specified character is plotted on the screen with its lower left hand corner at the specified coordinate, using the current graphics foreground colour and action.

See the technical description in this chapter for a description of plot actions.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 52 (SWI &2E)

Put sprite scaled

On entry

R0 = 52 (&34)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate to plot
R4 = y coordinate to plot
R5 = plot action
R6 = scale factors
R7 = pixel translation table

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved
R5 preserved
R6 preserved
R7 preserved

Use

This will plot a sprite on the screen using:

- the coordinate specified by R3 and R4
- the plot action specified by R5.
- the scale factor specified by R6
- the pixel translation table pointed to by R7

The plot actions specified in R5 are:

Value	Action
0	Overwrite colour on screen
1	OR with colour on screen
2	AND with colour on screen
3	exclusive OR with colour on screen
4	Invert colour on screen

5	Leave colour on screen unchanged
6	AND with colour on screen with NOT of sprite pixel colour
7	OR with colour on screen with NOT of sprite pixel colour
&08	If set, then use the mask, otherwise don't
&10	ECF pattern 1
&20	ECF pattern 2
&30	ECF pattern 3
&40	ECF pattern 4
&50	Giant ECF pattern (patterns 1 - 4 placed side by side)

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 53 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 53 (SWI &2E)

Put sprite grey scaled

On entry

R0 = 53 (&35)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate to plot at
R4 = y coordinate to plot at
R5 = 0
R6 = scale factors
R7 = pixel translation table

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved
R4 preserved
R5 preserved
R6 preserved
R7 preserved

Use

This call is similar to OS_SpriteOp 52, except that it performs anti-aliasing on the sprite as it scales it. This is the same technique that the Font Manager uses on characters. This means that the sprite must have been defined in a 4 bits per pixel mode (16 colours), and the pixels must reflect a linear grey scale, as with anti-aliased font definitions.

This call is considerably slower than OS_SpriteOp 52 (Put sprite scaled) and should only be used when the quality of the image is of the utmost importance. To speed up redrawing of an anti-aliased sprite, it is possible to draw the image into another sprite (using OS_SpriteOp 60 – switch output to sprite), which can then be redrawn more quickly.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 52 (SWI &2E)

Related vectors	SpriteV

OS_SpriteOp 54 (SWI &2E)

Remove left hand wastage

On entry

R0 = 54 (&36)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 preserved
R1 preserved
R2 preserved

Use

In general, sprites have a number of unused bits in the words corresponding to the left and right hand edges of each pixel row. This call removes the left hand wastage, so that the left hand side of the sprite is word aligned.

The right hand wastage is increased by the number of bits that were removed. If this is now more than 32 bits then a whole word is removed from each row of the sprite, and the rest of the sprite area moved down to fill the gap.

Note that when you switch output to a sprite using OS_SpriteOp 60 or 61, the left-hand wastage is also removed.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 60 and 61 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 60 (SWI &2E)

Switch output to sprite

On entry

R0 = 60 (&3C)
R1 = pointer to control block of sprite area
R2 = sprite pointer to switch to sprite or 0 to switch to screen
R3 = save area
- 0 = no save area
- 1 = system save area
- any other value = pointer to save area

On exit

R0 set to previous values
R1 set to previous values
R2 set to previous values
R3 set to previous values

Use

This call can cause VDU calls to be sent to the screen memory, or to a sprite's image.

R2 has its usual function as a sprite pointer or it can be zero. If it is a sprite pointer, then this call will switch VDU output to a sprite. If it is a zero, then this call will switch output to the screen.

The save area can have a number of values. If it is zero, then no save area will be used. If it is one, then the system save area is used, which is the save area used by RISC OS when output is directed to the screen. You should not use the system save area yourself if you wish to preserve the VDU output state for the screen. Any other value of R3 is considered to be a pointer to the save area.

If the first word of the save area is zero, then the VDU state will be initialised to suitable defaults for the given sprite's mode. When output is switched away from the sprite, the current VDU state is copied into the save area, and the first word is overwrittenwith a non-zero value. If output is subsequently switched back to the sprite, the VDU state will be restored from the save area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 61 and 62 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 61 (SWI &2E)

Switch output to mask

On entry

R0 = 61 (&3D)
R1 = pointer to control block of sprite area
R2 = sprite pointer to switch to mask or 0 to switch to screen
R3 = save area

0 = no save area
1 = system save area
any other value = pointer to save area

On exit

R0 set to previous values
R1 set to previous values
R2 set to previous values
R3 set to previous values

Use

This call can cause VDU calls to be sent to the screen memory, or to a sprite's mask.

A sprite's mask has the same number of bits per pixel as its image, where a value of 0 is a transparent pixel and a value of all 1's represents a solid pixel. For example, &0F for 4 bits per pixel. Other values are not permitted.

See OS_SpriteOp 60 for a general description of how this call works.

Note that when plotting into a sprite's mask, the only colours that should be used are 0 and (number of colours –1), that is:

- in 2 colour modes use colours 0 and 1
- in 4 colour modes use colours 0 and 3
- in 16 colour modes use colours 0 and 15
- in 256 colour modes use colour 0 tint 0, and colour 63 tint 192 (&C0)

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 60 and 62 (SWI &2E)

Related vectors	SpriteV

OS_SpriteOp 62 (SWI &2E)

Read save area size

On entry

R0 = 62 (&3E)
R1 = pointer to control block of sprite area
R2 = sprite pointer, or 0 for the screen

On exit

R0 preserved
R1 preserved
R2 preserved
R3 = size of required save area in bytes

Use

This calls calculates how large a save area must be for a given sprite. Remember that a save area must be word aligned.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – see the start of the earlier section on *Technical Details* for a description.

Related SWIs

OS_SpriteOp 60 and 61 (SWI &2E)

Related vectors

SpriteV

*Configure SpriteSize

Reserve a given amount of memory for the system sprite area

Syntax

```
*Configure SpriteSize <n>[K]
```

Parameters

`<n>`	number of pages of memory; n<=127
or	
`<n>K`	number of kilobytes of memory reserved

Use

`*Configure SpriteSize` is used to reserve an area of memory for the system sprites. If n=0, then no space is reserved for system sprites. The default value is one page of memory.

As with all configures, this command does not come into effect until the next hard break.

You can also use OS_ChangeDynamicArea (SWI &2A) to alter the system sprite size at runtime. For more information, refer to the chapter entitled *Memory Management*.

Example

```
*Configure SpriteSize 20K
```

Related commands

None

Related SWIs

None

Related vectors

None

*SChoose

Select a sprite

Syntax	`*SChoose <name>`
Parameters	`<name>` name of the sprite in the system sprite area
Use	`*SChoose` selects a particular sprite for use in subsequent sprite plotting operations. That is, it is used in conjunction with VDU 25,232-239 operations. See the note in the Technical Details about using this command. The sprite names are not case-sensitive.
Example	`*SChoose fish`
Related commands	None
Related SWIs	OS_SpriteOp 24 (SWI &2E)
Related vectors	SpriteV

*SCopy

Copy a sprite

Syntax	`*SCopy <name1> <name2>`
Parameters	`<name1>` name of the original sprite `<name2>` name of the new copy both are in the system sprite area
Use	`*SCopy` makes a copy of `name1` and renames it `name2`. An error will occur if `name2` already exists.
Example	`*SCopy acorn squirrel`
Related commands	None
Related SWIs	OS_SpriteOp 27 (SWI &2E)
Related vectors	SpriteV

*ScreenLoad

Load a single sprite file into the graphics window

Syntax

`*ScreenLoad <pathname>`

Parameters

`<pathname>`	name of file to load

Use

`*ScreenLoad` plots a sprite directly from a file into the graphics window. It changes mode if necessary and sets the palette to the setting in the file. The first sprite in the file is plotted at the bottom left hand corner of the graphics window. After a mode change, this is the bottom left hand corner of the screen.

Example

```
*ScreenLoad $.sprites.animals.koala
```

Related commands

*ScreenSave

Related SWIs

OS_SpriteOp 3 (SWI &2E)

Related vectors

SpriteV

*ScreenSave

Save the current graphics window and palette as a sprite file

Syntax

`*ScreenSave <pathname>`

Parameters

`<pathname>`	full name of file to save

Use

`*ScreenSave` saves the contents of the screen bounded by the current graphics window, along with the current palette, into a file. The sprite file created will contain one sprite called 'screendump'.

Example

`*ScreenSave $.sprites.animals.koala`

Related commands

*ScreenSave

Related SWIs

OS_SpriteOp 2 (SWI &2E)

Related vectors

SpriteV

*SDelete

Delete a sprite

Syntax

`*SDelete <name1> [<name2>...]`

Parameters

`<name1>`	name of sprite in the system sprite area to delete
`<name2>...`	optional extra sprites to delete

Use

`*SDelete` deletes one or more sprites from the system sprite area.

`*SDelete` will stop immediately, following an error.

Example

```
*SDelete fish cake elephant
```

Related commands

None

Related SWIs

OS_SpriteOp 25 (SWI &2E)

Related vectors

SpriteV

*SFlipX

Reflect a sprite about its x axis

Syntax

`*SFlipX <name>`

Parameters

`<name>`	name of the sprite in the system sprite area

Use

`*SFlipX` reflects the named sprite in the system area about its x axis so it is upside down.

Example

```
*SFlipX sloth
```

Related commands

*SFlipY

Related SWIs

OS_SpriteOp 33 (SWI &2E)

Related vectors

SpriteV

*SFlipY

Reflect a sprite about its y axis

Syntax	`*SFlipY <name>`
Parameters	`<name>` name of the sprite in the system sprite area
Use	`*SFlipY` reflects the named sprite in the system area about its y axis so it faces in the opposite direction.
Example	`*SFlipY sloth`
Related commands	*SFlipX
Related SWIs	OS_SpriteOp 47 (SWI &2E)
Related vectors	SpriteV

*SGet

Get a sprite from the screen

Syntax	`*SGet <name>`
Parameters	`<name>` name of new sprite in the system sprite area
Use	`*SGet` picks up a rectangular area of the screen, defined by the two most recent graphics positions (inclusive). It then saves this as a sprite in the system area under the name given. If the sprite already exists, it is overwritten. Any part of the designated area which lies outside the current graphics window is filled with the current background colour in the sprite.
Example	`*SGet screenpart`
Related commands	*ScreenSave
Related SWIs	OS_SpriteOp 14 (SWI &2E)
Related vectors	SpriteV

*SInfo

Get information on the system sprite workspace

Syntax

```
*SInfo
```

Parameters

None

Use

`*SInfo` prints out the amount of system sprite workspace currently reserved, the amount of free space in that workspace and the number of sprites defined.

Example

```
*SInfo
Sprite status
  8 Kbytes sprite workspace
  7328 byte(s) free
  2 sprite(s) defined
```

Related commands

None

Related SWIs

OS_SpriteOp 8 (SWI &2E)

Related vectors

SpriteV

*SList

List all the system sprites

Syntax	`*SList`
Parameters	None
Use	`*SList` prints a list of the names of all the sprites in the system sprite area.

Example	`*SList`
	`!koala`
	`!sloth`

Related commands	None
Related SWIs	OS_SpriteOp 8 (SWI &2E)
Related vectors	SpriteV

*SLoad

Load a sprite file into the system sprite area

Syntax	`*SLoad <pathname>`
Parameters	`<pathname>` full name of file to load
Use	`*SLoad` loads a file containing sprite definitions into the system sprite area. If there is insufficient memory, then an error is given and nothing is loaded. Any sprites which are in memory when this command is given are lost.
Example	`*SLoad $.sprites.animals.koala`
Related commands	*ScreenLoad
Related SWIs	OS_SpriteOp 10 (SWI &2E)
Related vectors	SpriteV

*SMerge

Merge a sprite file with those in the system sprite area

Syntax

`*SMerge <pathname>`

Parameters

`<pathname>`	full name of file to load

Use

`*SMerge` merges the sprites in a file with those in the system sprite area. If there is insufficient memory, then an error is given and nothing is loaded. Any sprites in memory with the same name as any in the file are lost.

Note that there must be enough free space in the sprite area to hold both the new file and the original sprites, since it is only after the new file has been loaded that any of the original sprites are replaced by new ones that have the same name.

Example

`*SMerge $.sprites.animals.koala`

Related commands

None

Related SWIs

OS_SpriteOp 11 (SWI &2E)

Related vectors

SpriteV

*SNew

Delete all the system sprites

Syntax

`*SNew`

Parameters

None

Use

`*SNew` deletes all the sprites in the system sprite area, and so frees all the sprite workspace.

Related commands

None

Related SWIs

OS_SpriteOp 9 (SWI &2E)

Related vectors

SpriteV

*SRename

Rename a sprite

Syntax	`*SRename <name1> <name2>`
Parameters	`<name1>` name of the original sprite `<name2>` new name of the sprite
Use	`*SRename` assigns the new name `name2` to the sprite currently called `name1`. `name2` must not already exist, or an error will occur A sprite name can contain any sequence of printable characters, other than a space; although upper-case letters will be changed to lower-case ones.
Example	`*SRename thong flipflop`
Related commands	None
Related SWIs	OS_SpriteOp 26 (SWI &2E)
Related vectors	SpriteV

*SSave

Save the system sprite area as a sprite file

Syntax	`*SSave <pathname>`
Parameters	`<pathname>` name of file to save
Use	`*SSave` saves all the sprites currently in the system sprite area to a file which can later be loaded or merged.
Example	`*SSave $.sprites.animals.koala`
Related commands	*SLoad, *SMerge
Related SWIs	OS_SpriteOp 12 (SWI &2E)
Related vectors	SpriteV