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.

Part 1

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

We've laid out the information in these parts 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 on which you can base your own programs.

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.

Indexes

The separate volume of Indexes 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
- an index by subject.

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.

`Bold Courier` type is used in some examples to show input from the user. We only use it where we need to distinguish between user input and computer output.

Command syntax

Special symbols are used when defining the syntax for commands:

- Italics indicate that you must substitute an actual value. For example, *`filename`* means that you must supply an actual filename.
- Braces indicates that the item enclosed is optional. For example, `[K]` shows that you may omit the letter 'K'.
- A bar indicates an option. For example, `0|1` means that you must supply the value 0 or 1.

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 define the SWI names as manifests for the SWI numbers. See the chapter entitled *An introduction to SWIs* on page 1-23
- 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 *RISC OS User Guide* and *RISC OS Applications Guide* supplied with it.

If you wish to write BASIC programs on your RISC OS computer you will find the *BBC BASIC Reference Manual* useful.

Your Acorn supplier has available the *Acorn Desktop C* and *Acorn Desktop Assembler* products, which you can use to write programs in (respectively) C and ARM assembler. Both products run in a desktop environment with full supporting tools. The manuals for both products are available separately if required: they are entitled *Acorn ANSI C Release 4* and *Acorn Assembler Release 2*.

Reader comments

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

Part 1 – Introduction

1 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*:

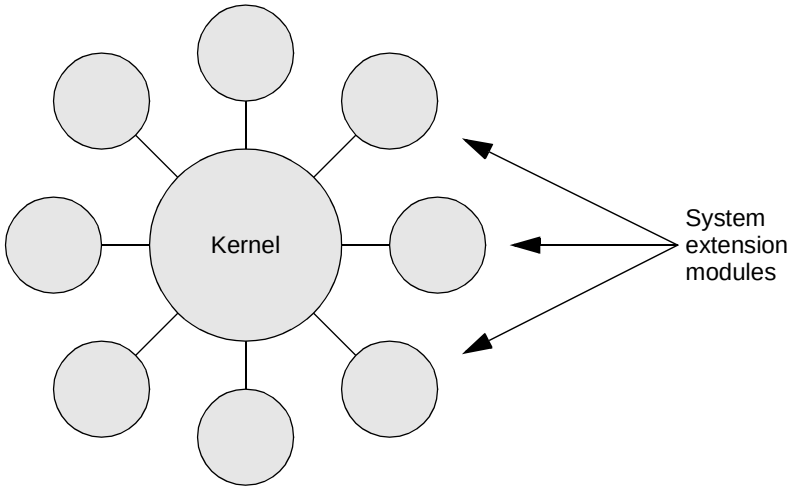


Figure 1.1 The structure of RISC OS

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 *SWI*s 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* on page 1-201 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 as 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 claiming just 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.

For more information, see the chapter entitled *Software vectors* on page 1-63.

How RISC OS is written

Much of RISC OS – including the kernel – is written in ARM assembler. Some other parts – such as the Filer_Action system extension module – are written in C, and so need the *Shared C Library* to work.

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

To use RISC OS effectively, it helps to have a working knowledge of the ARM processor and of ARM assembler yourself. The chapter entitled *ARM Hardware* on page 1-9 provides a brief introduction to the ARM processor and the set of chips that support it. The appendix entitled *Appendix A: ARM assembler* on page 4-363 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.

The history of RISC OS

This manual describes RISC OS 3, which was developed from RISC OS 2. This in turn derives 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.

Two different versions of RISC OS 2 were released:

- RISC OS 2.00 was the original release
- RISC OS 2.01 was a later release which added support for the Archimedes 500 series machines; it was not fitted to other machines.

The differences between these two versions are so few, that **unless we need to differentiate between them** we shall refer to them both as 'RISC OS 2'.

There are currently three different versions of RISC OS 3:

- Version 3.00 was the initial release, made for the A5000 computer.
- Version 3.10 was a considerably improved release that added support for other Acorn computers, including both older RISC OS computers, and the new A4, A3010, A3020 and A4000 computers.
- Version 3.11 has very minor differences from 3.10, but its programmer's interfaces are exactly the same, and you can treat it as identical.

Again, unless we need to differentiate between versions, we shall refer to them all as 'RISC OS 3'.

Arthur

There are very few remaining users of Arthur, and we consider it to be obsolete. You should not worry about making your programs compatible with Arthur.

In view of this, we do not distinguish features and facilities that are available under RISC OS but not under Arthur. However, you will find most of the facilities of Arthur described in this manual, because they have been subsumed into RISC OS. If you need **full** details of how Arthur did things, so you can maintain old programs, you'll have to refer to the *Programmers Reference Manual* that was released with Arthur. Don't throw your old manuals away – keep them!

Some minor parts of the Arthur operating system, which were in the *Programmers Reference Manual* released with Arthur, are not in this manual. This is because we now consider them to be obsolete, even though they're generally still supported. Instead, we've documented the preferred way of getting the same results under RISC OS. Likewise, some other parts of Arthur are only referred to in passing.

RISC OS 2 documentation

Because some users may prefer not to upgrade from RISC OS 2 to RISC OS 3, we advise you to write applications so that they will still run under both versions. This will maximise your potential market with very little extra effort. To help you in this, we say explicitly whenever a facility or feature is specific to a version of RISC OS.

We've derived this manual directly from the *RISC OS Programmer's Reference Manual* written for RISC OS 2. Any changes or additions you notice have been made for one of these reasons:

- To cover a change or addition to RISC OS.

  In such cases this is explicitly stated, together with information on the versions of RISC OS to which the change or addition are applicable.

- To improve the clarity and accuracy of the original *RISC OS Programmer's Reference Manual*, or to correct an error.

  Such improvements and corrections are not explicitly identified. You may assume that, where this manual differs from the previous edition, it is this later edition that is correct.

2 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 current 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.*

Some machines combine the functionality of one or more of these chips as macrocells on a single chip: for example the A3010 and A3020.

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 400 series computer, which may be viewed as typical of a RISC OS computer:

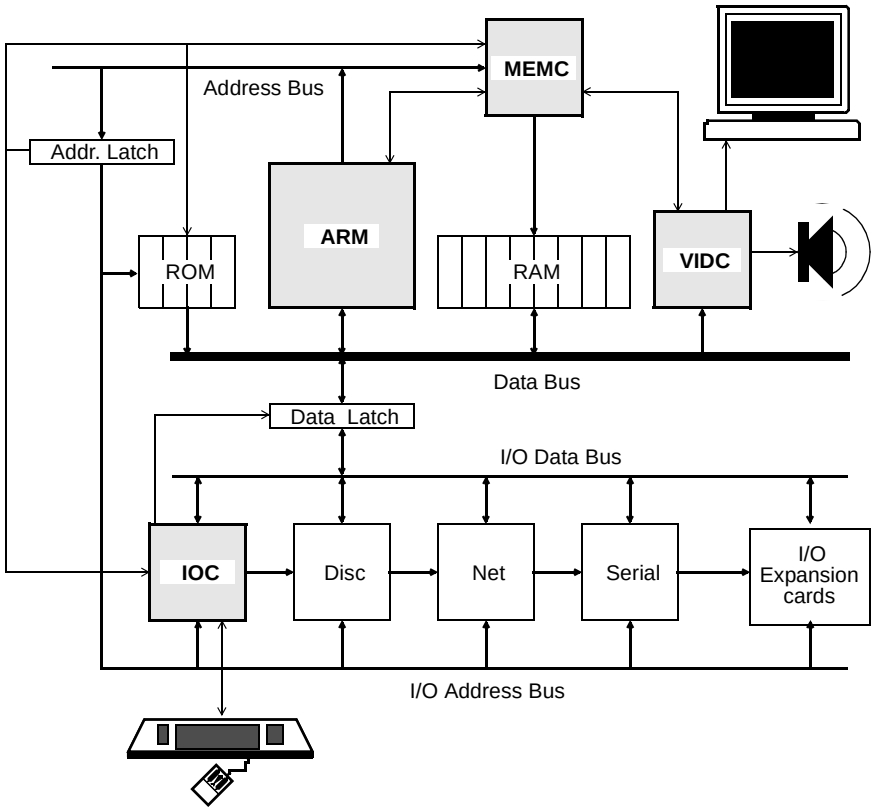


Figure 2.1 Architecture of an Archimedes 400 series computer

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 used for 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.

Advantages of RISC

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

- it has a high performance
- it uses much less power than comparable CISC chips
- 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.

ARM 2 and ARM 3

Currently Acorn uses two different versions of the ARM processor. The newer ARM 3 is clocked at about three times the speed of the older ARM 2, and has a 4Kbyte on-chip cache. These two features mean that it delivers some three times the power of the ARM 2 (13.5 million instructions per second, or MIPS, compared to some 4 - 5 MIPS for the ARM 2).

From the programmer's point of view, there is little difference between the two processors. The ARM 3 supports the full instruction set of the ARM 2, and provides a few extra instructions: all but one of these instructions are used to control the ARM 3's cache.

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.

This is known as *pipelining*.

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 `MOV 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 light grey:

<table>
<tr><th>User Mode</th><th>SVC Mode</th><th>IRQ Mode</th><th>FIQ Mode</th></tr>
<tr><td colspan="4">R0</td></tr>
<tr><td colspan="4">R1…R6</td></tr>
<tr><td colspan="4">R7</td></tr>
<tr><td colspan="3">R8</td><td>R8_fiq</td></tr>
<tr><td colspan="3">R9</td><td>R9_fiq</td></tr>
<tr><td colspan="3">R10</td><td>R10_fiq</td></tr>
<tr><td colspan="3">R11</td><td>R11_fiq</td></tr>
<tr><td colspan="3">R12</td><td>R12_fiq</td></tr>
<tr><td>R13</td><td>R13_svc</td><td>R13_irq</td><td>R13_fiq</td></tr>
<tr><td>R14</td><td>R14_svc</td><td>R14_irq</td><td>R14_fiq</td></tr>
<tr><td colspan="4">R15 (PC/PSR)</td></tr>
</table>

Figure 2.2 ARM registers

Only two of the registers have special functions:

- R15 is used as the program counter (*PC*) and processor status register (*PSR*)
- R14 (and R14_svc, R14_irq, R14_fiq) 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:

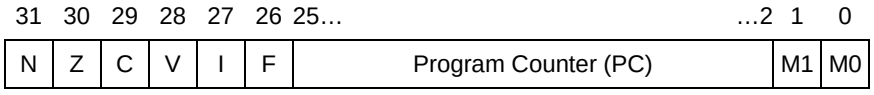


Figure 2.3 Bit usage in R15 by the PC and the PSR

- 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 o**V**erflow flag
  C **C**arry flag
  Z **Z**ero flag
  N **N**egative flag

The program counter must always be word aligned, and so the lowest two bits of the address must always be zero. To maximise the available address space, these two bits are not stored in R15, but are appended to the program counter when fetching instructions, thus forming a 26-bit address.

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) 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 the appendix entitled *Appendix A: ARM assembler* on page 4-363.

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 its 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 a FIFO (first-in, first-out). 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 $^1/_3$, $^1/_2$, $^2/_3$ or 1 times the clock rate of VIDC (on older machines this is always 24MHz, but on newer machines you may also select clock rates of 25.175 or 36MHz)
- 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
- the width of the screen border can be altered.

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). Its pixels can be transparent (so the cursor can be any shape you desire), or can use any three of the 4096 possible colours.

The cursor may be positioned anywhere on the screen within the border.

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/Output (*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 peripheral controllers that require an interruptible 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 200ns where MEMC is clocked at 8MHz, and lower (in inverse proportion) where MEMC is clocked at 10 or 12MHz.

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

MEMC maps the physical memory into a 16 Mbyte slot, the base of which is at 32 Mbytes. RISC OS does not address this slot 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 the slot that RISC OS addresses can be:

- unmapped
- mapped onto one page of the physical memory
- mapped onto many pages of the physical 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. These pages are, of course, not held at &8000.

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 (or CAM). 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.

MEMC addresses a maximum of 4 Mbytes of memory. Machines with more than 4 Mbytes fitted have an extra MEMC chip slaved to the master MEMC chip for each additional 4 Mbyte fitted, so the page sizes are the same as for a 4 Mbyte machine.

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 per MEMC chip, 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
8 Mbytes	32 Kbytes	1 K
16 Mbytes	32 Kbytes	1 K

Figure 2.4 Page sizes for Archimedes computers

If you need to find out a machine's page size and so on, use OS_ReadMemMapInfo (page 1-391).

Number of pages programmed

RISC OS programs a minimum of 128 pages, even if it actually uses fewer pages. This is so that:

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

Protection modes and levels

MEMC can run in three different protection modes: Supervisor, Operating System, and User. Each page of memory has one of three different protection levels: 0, 1 and 2. (There is also a level 3, which is identical to level 2.) Whether you have read and/or write access to a page of memory is determined by MEMC's current protection mode, and by the protection level of that page:

- Supervisor mode is the most privileged mode, adopted whenever the ARM is in one of its privileged modes (SVC, IRQ or FIQ). It gives read/write access to pages of all protection levels, so the whole address space can be freely accessed.
- Operating System mode is entered by setting a bit in MEMC's control register, which can only be altered if the ARM is in a privileged mode (although this bit remains set when the ARM leaves such a mode). It allows read/write access to pages of protection level 0 or 1, and read-only access to pages of protection level 2. The page protection levels under RISC OS are such that it it is a more privileged mode than User mode when accessing logically mapped RAM, but acts as User mode in all other cases.

  RISC OS itself does not use Operating System mode.
- User mode is the least privileged mode, adopted in all other circumstances. It allows read/write access to pages of protection level 0, read-only access to pages of protection level 1, and no access to pages of protection level 2. Under RISC OS it allows read/write access only to unprotected pages in the logically mapped RAM, and read access 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.

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:

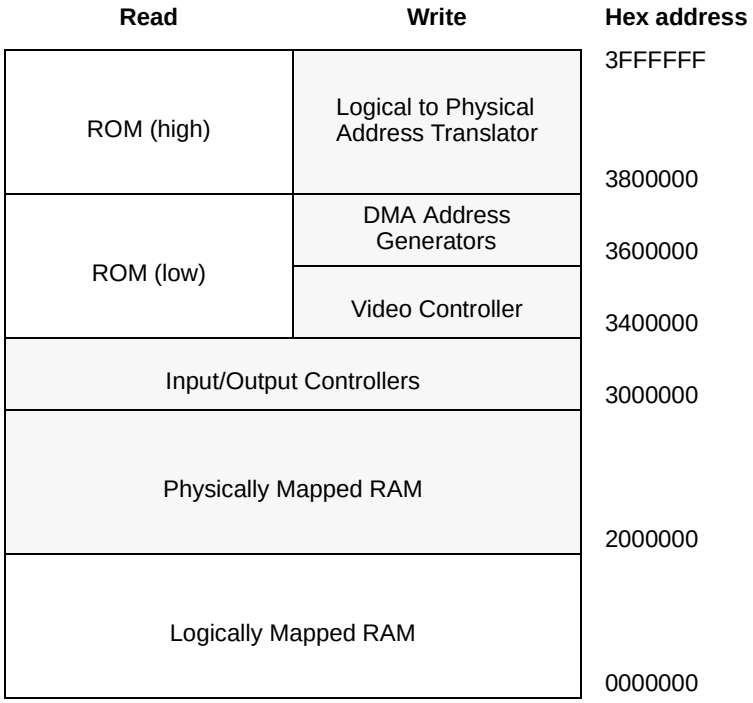


Figure 2.5 Memory map of a typical RISC OS computer

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 the appendix entitled *Appendix A: ARM assembler* on page 4-363
- ARM assembler is thoroughly covered in the manual supplied with the *Desktop Assembler*, available from your Acorn supplier
- The ARM chip set is described in much greater detail in the *Acorn RISC Machine family Data Manual*. VLSI Technology Inc. (1990) Prentice-Hall, Englewood Cliffs, NJ, USA: ISBN 0-13-781618-9.

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

3 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 **S**oft**W**are **I**nterrupt, 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 normally 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 will 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 uniquely identifies a routine. This field is known as the *SWI number*. The section entitled *SWI numbers in detail* on page 1-26 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.

- SWIs may use registers R0 - R9 inclusive.
- Note that the mechanism for calling SWIs from BASIC will only handle registers R0 - R7 inclusive. For this reason, parameters are normally restricted to these registers.

Fortunately it is rare that a routine needs 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 Reference Manual*. 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* on page 1-41.

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 - &3FFF (262143) provide the basic operating system functions. (Only a small proportion 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', or 'UKSWIV' for short. 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		Meaning
19	18	
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* on page 1-41.

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:

000010nnnnnnnnnnnn000000 to
000010nnnnnnnnnnnn111111

where nnnnnnnnnnnn 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 normally 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* on page 1-41.)

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* on page 1-41 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 instruction is decoded, and so on.

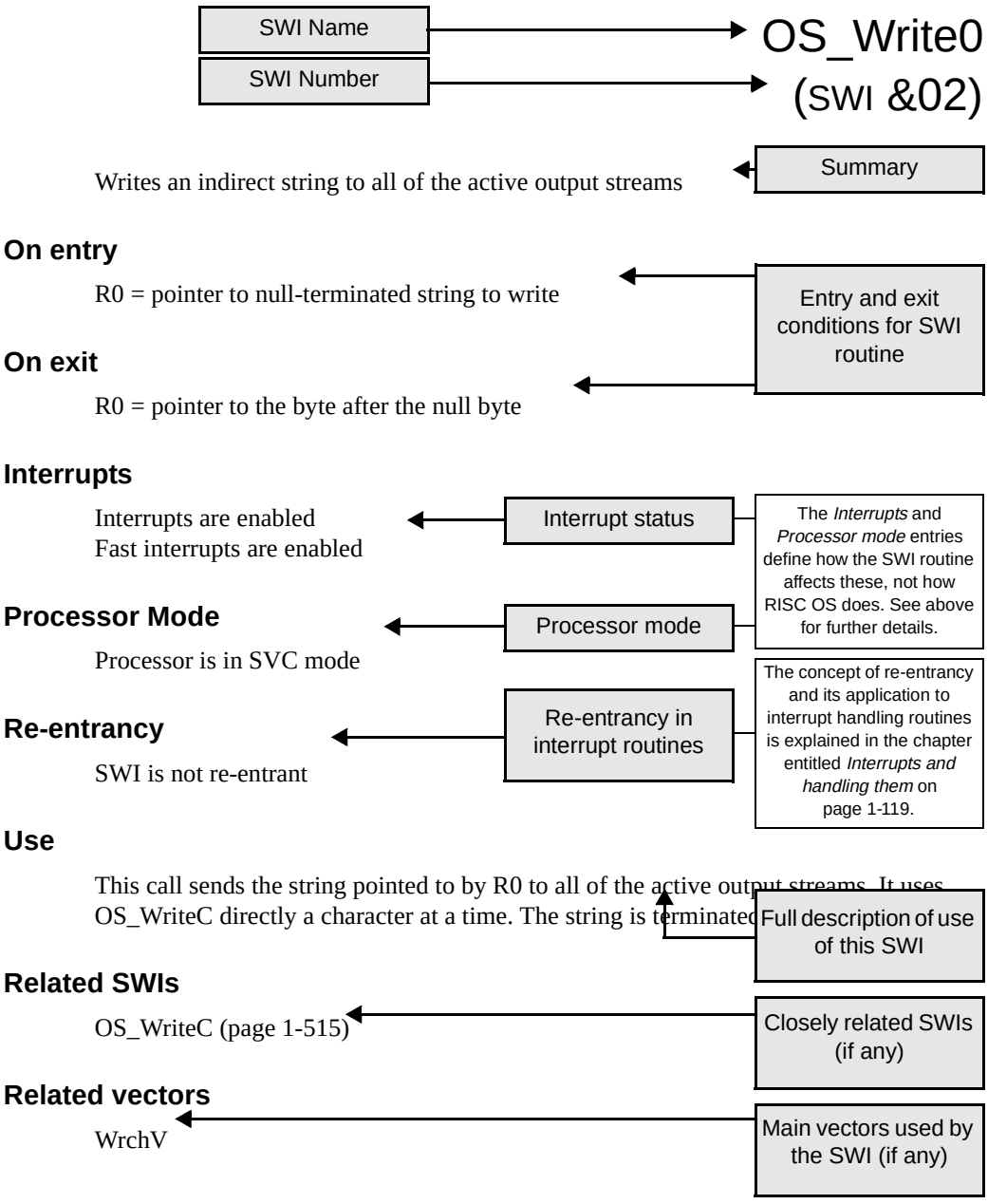


Figure 3.1 Example SWI documentation

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 in succession, 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.

4 * 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 *RISC OS User Guide*, you may already be familiar with many of these commands.

This chapter introduces you to * Commands and the CLI; the chapter entitled *The CLI* on page 1-955 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* on page 1-451. You will find many of the miscellaneous * Commands that the kernel supplies in the chapter entitled *The CLI* on page 1-955. (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:

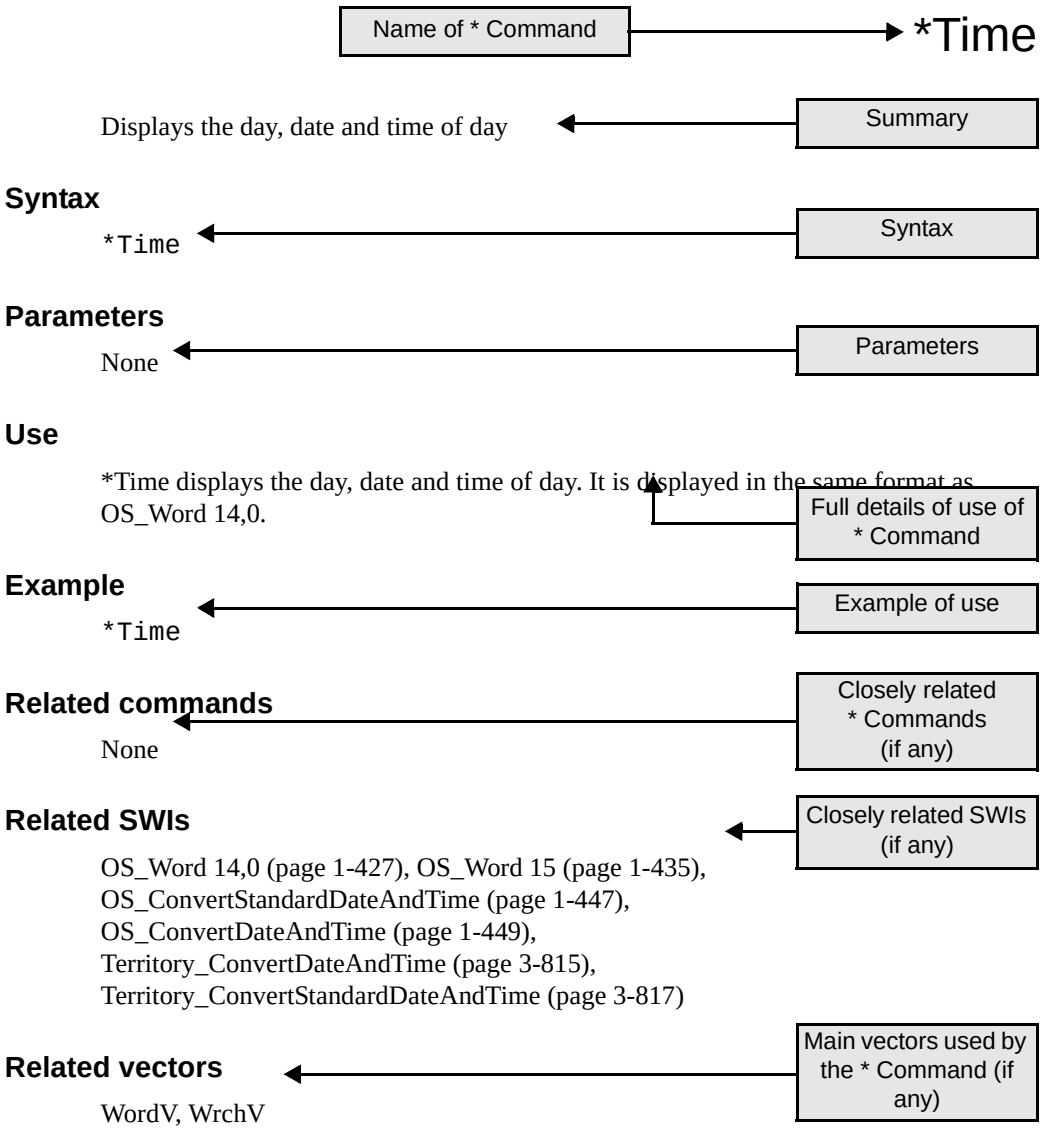


*Figure 4.1 Example * Command documentation*

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 with command lines

You can issue * Commands from most old-fashioned applications that provide a command line 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.

Should you write an application that provides a command line (which we deprecate in favour of the desktop), it 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 null byte terminating the command string:

```
                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 section entitled *Calling from BBC BASIC* on page 1-24. 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
                EQUB 0                  ; Terminating null for text
                ALIGN
```

See the *BBC BASIC Reference Manual* 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 can run a replacement application using this call by prefixing it with '`CHAIN: `'. So:

```
system("CHAIN: BASIC")
```

would start up BBC BASIC, returning control to the C application's parent when BASIC quits, whereas:

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

starts 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* on page 1-63 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* on page 1-201 for details of how to write a module.

Using Alias$*command* system variables

You can use system variables of the form Alias$*command* to create new commands from existing ones, or to rename existing commands. For more details, see the section entitled *Aliases* on page 1-958, the command **Set* on page 1-331, and the section entitled *Changing and adding commands* on page 4-503.

5 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 must 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 - &19F	OS_Heap errors
&1A0 - &1AF	OS_Claim/Release errors
&1B0 - &1BF	OS_ChangeEnvironment errors
&1C0 - &1DF	OS_ChangeDynamicArea 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 - &87F	Debugger errors
&880 - &8FF	BBC I/O Podule errors
&900 - &97F	Shell CLI errors, and miscellaneous others
&980 - &9FF	Draw errors
&A00 - &A3F	ColourTrans errors

&A40 - &A7F	ARM3 errors
&A80 - &ABF	TaskWindow errors
&AC0 - &AFF	MessageTrans errors
&B00 - &B3F	Pinboard errors
&B40 - &B4F	Portable module errors
&1XX00 - &1XXFF	Errors from filing system number &XX
(eg &10800 - &108FF	ADFS errors)
&20000 - &200FF	Sound errors
&20200 - &21000	Podule errors, and miscellaneous others

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 (24 - 29) 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 (for a detailed description see the section entitled *ErrorV* on page 1-80). 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 entitled *Character Input* on page 1-863 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.

SWI Calls

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_GenerateError (SWI &2B) **or**
V flag is set – XOS_GenerateError (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
   BLCS  acknowledge_escape             ; acknowledge it - returns using
                                        ; LDMFD R13!,{PC}^
   ADRCS R0,escapeBlock                 ; Get ptr. to error block
   MOVCS R1,#0                          ; Use global messages file
   MOVCS R2,#0                          ; Use internal buffer
   SWICS "XMessageTrans_ErrorLookup"    ; Look up token in message file
   SWICS "OS_GenerateError"             ; Do the error – doesn't return
.noEscape
   ...
.escapeBlock
   EQUD  17                             ; Error number for escape
   EQUS  "Escape"+CHR$0                 ; Error token to lookup
   ALIGN
```

Related SWIs

None

Related vectors

ErrorV

* Commands

*Error

Generates errors

Syntax

`*Error [error_no] text`

Parameters

error_no	the error number
text	a string of printable characters explaining the error

Use

*Error generates an error with the given error number and explanatory text. This is normally then printed on the screen. This command is useful for reporting errors after trapping them within a command script.

If you omit the error number it is set to the default value of 0.

Errors are also, of course, generated by RISC OS itself.

Example

```
*Error 100 No such file
```
prints 'No such file'

Related commands

None

Related SWIs

OS_GenerateError (SWI &2B)

Related vectors

ErrorV

6 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 older 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 *Index of OS_Bytes* on page Index-13.

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 a routine to implement or decode OS_Byte calls, 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 temporary registers 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* on page 1-23 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 (inclusive)

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 (inclusive)

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* on page 1-63.

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* on page 1-201.

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,#138
MOV     R1,#0
MOV     R2,#65
SWI     OS_Byte
```

has the same effect as:

```
*FX 138,0,65
```

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* on page 1-33 for details.

SWI calls

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 (inclusive), 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 (inclusive) 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.

For an index of all OS_Byte calls, see the *Index of OS_Bytes* on page Index-13.

Related SWIs

OS_Word (page 1-61)

Related vectors

ByteV

* Commands

*FX

Calls OS_Byte to alter status variables, and to perform other closely related actions

Syntax

`*FX` *`reason_code`* `[[,]` *`r1`* `[[,]` *`r2`*`]]`

Parameters

`reason_code`	from 0 to 255
`r1`	from 0 to 255
`r2`	from 0 to 255

The parameters are in decimal by default, but you may specify other bases (see *Examples* below).

Use

*FX alters status variables, which the computer uses to control its operation. You can either read from them, or write to them. Some *FX commands will also perform other actions closely related to the status variable that is being altered.

This command merely calls the SWI OS_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 138, … will do, see the entry for OS_Byte 138. For an index of all OS_Byte calls, see the *Index of OS_Bytes* on page Index-13.

Examples

`*FX 138,0,&41`	*r2 is specified in hexadecimal*
`*FX 247 4_01`	*r1 is specified in base 4*

Related commands

None

Related SWIs

OS_Byte (page 1-54)

Related vectors

ByteV

7 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* on page 1-63. Alternatively you can use the Unknown OS_Word service call, although this is not such a good way to do so, and is hence deprecated – see the chapter entitled *Modules* on page 1-201.

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. For details see the earlier chapter entitled *An introduction to SWIs* on page 1-23.

OS_Word and * Commands

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

SWI calls

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. For an index of all OS_Word calls, see the *Index of OS_Words* on page Index-17

Related SWIs

OS_Byte (page 1-54)

Related vectors

WordV

8 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 workspace 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 identical earlier instances of the routine from the chain (ie instances having the same pointer to code, and the same pointer to workspace). If you don't want this to happen, use the SWI OS_AddToVector instead.

Desktop applications

Under an environment such as the desktop, multiple applications are run concurrently. The currently running application is mapped into memory at &8000. Desktop applications periodically return control to the Window Manager (or *Wimp*) by calling the SWI Wimp_Poll; at this point the Wimp may decide to swap to another application. In doing so, it maps the current application out of the application space, and maps the new application into that space. Thus every application is given the illusion that it is the only one in the system.

If your application has claimed a vector using a routine in its own space, it must obviously release that vector each time it (and the claiming routine) may be swapped out of application space. Before each call your application makes to Wimp_Poll (which is when it may be swapped out), it must call OS_DelinkApplication (page 1-74) to remove any claiming routines in application space. When its call to Wimp_Poll returns (and hence it is swapped back in), it must then call OS_RelinkApplication (page 1-76) to reclaim those vectors.

SWI Calls

OS_Claim (SWI &1F)

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

On entry

R0 = vector number (see page 1-78)
R1 = address of claiming routine that is to be added to vector
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 identical earlier instances of the routine are removed. Routines are defined to be identical 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.

Example

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

Related SWIs

OS_Release (page 1-68), OS_CallAVector (page 1-70), OS_AddToVector (page 1-72)

Related vectors

All

OS_Release (SWI &20)

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

On entry

R0 = vector number (see page 1-78)
R1 = address of routine that is to be released from vector
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 call 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. If more than one copy of the routine is claiming the vector, only the first one to be called is removed.

Example

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

Related SWIs

OS_Claim (page 1-66), OS_CallAVector (page 1-70), OS_AddToVector (page 1-72)

Related vectors

All

OS_CallAVector (SWI &34)

Calls a vector directly

On entry

R0 - R8 = vector routine parameters
R9 = vector number (see page 1-78)
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 to 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 (page 1-66), OS_Release (page 1-68), OS_AddToVector (page 1-72)

Related vectors

All

OS_AddToVector (SWI &47)

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

On entry

R0 = vector number (see page 1-78)
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.

Related SWIs

OS_Claim (page 1-66), OS_Release (page 1-68), OS_CallAVector (page 1-70)

Related vectors

All

OS_DelinkApplication (SWI &4D)

Remove any vectors that an application is using

On entry

R0 = pointer to buffer
R1 = buffer size in bytes

On exit

R0 preserved
R1 = number of bytes left in buffer

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entrant because interrupts are disabled

Use

When an application running at &8000 is going to be swapped out, it must remove all vectors that it uses. Otherwise, if they were activated, they would jump into whatever happened to be at that location in the new application running in that space.

R0 on entry points to a buffer. This is used to store details of the vectors used, so that they can be restored afterwards. Each vector requires 12 bytes of storage and the list is terminated by a single byte.

If the space left returned in R1 is zero, then you must allocate another buffer and repeat the call; the buffer you have contains valid information. When you relink you must pass all the buffers returned by this call.

Related SWIs

OS_RelinkApplication (page 1-76)

Related vectors

None

OS_RelinkApplication (SWI &4E)

Restore from a buffer any vectors that an application is using

On entry

R0 = pointer to buffer

On exit

R0 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

When an application is going to be swapped in, all vectors that it uses must be restored. R0 on entry points to a buffer, which has previously been created by OS_DelinkApplication.

Related SWIs

OS_DelinkApplication (page 1-74)

Related vectors

None

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 section entitled *Important notes* on page 1-31 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 section entitled *Restrictions* on page 1-128.

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. These are described immediately below; for an example, see the section entitled *An example program* on page 1-107.

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
```

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 following section entitled *Summary of vectors* gives a summary of each vector, and tells you where to find out more about it. A few vectors also merit a more detailed description in the section entitled *Vector descriptions* on page 1-85. Such vectors are indicated in the list below by a dagger '†'. Also, the names of the routines which can cause each 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)
RdchV		(&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)
FSCV		(&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/Flush 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 vector
UpcallV		(&1D)	Warning vector (OS_UpCall)
ChangeEnvironmentV		(&1E)	Environment change vector (OS_ChangeEnvironment)
SpriteV		(&1F)	OS_SpriteOp indirection vector (OS_SpriteOp)
DrawV	†	(&20)	Draw SWI vector (Draw_…)
EconetV	†	(&21)	Econet activity vector (Econet_…)
ColourV	†	(&22)	ColourTrans SWI vector (ColourTrans_…)
PaletteV	†	(&23)	Read/write palette vector
SerialV		(&24)	OS_SerialOp indirection vector

All other vectors are reserved by Acorn.

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, starting on page 2-11) 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, and you must not use it. Its default action is to do nothing.

ErrorV

ErrorV is used to indirect all errors from error-generating SWIs and from OS_GenerateError – see page 1-45 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* on page 1-287 for more about handlers.

IrqV

IrqV is called when an unknown IRQ is detected. It was provided in earlier versions of RISC OS to enable you to add interrupt generating devices of your own to the computer, but is now considered obsolete. The default action is to disable the interrupting device. See page 1-86 later in this chapter for full details.

See also the chapter entitled *Interrupts and handling them* on page 1-119, and the chapter entitled *Program Environment* on page 1-287 for more about handlers.

WrchV

WrchV is used to indirect all calls to OS_WriteC – see page 1-515 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 page 1-880 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 page 1-961 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 page 1-54 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), and be prepared to be re-entered.

WordV

WordV is used to indirect all calls to OS_Word – see page 1-61 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), and be prepared to be re-entered.

FileV

FileV is used to indirect all calls to OS_File – see page 2-32 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 page 2-49 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 page 2-63 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 page 2-65 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 page 2-66 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 page 2-75 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 page 1-941 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 page 2-80 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 page 1-154 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* on page 1-287 for more about handlers.

InsV

InsV is called to place one or more bytes in a buffer. See page 1-88 later in this chapter for full details.

See also the chapter entitled *Buffers* on page 1-163.

RemV

RemV is called to remove one or more bytes from a buffer. See page 1-90 later in this chapter for full details.

See also the chapter entitled *Buffers* on page 1-163.

CnpV

CnpV is called to count the number of entries or spaces in a buffer, or to flush the contents of a buffer. See page 1-92 later in this chapter for full details.

See also the chapter entitled *Buffers* on page 1-163.

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 page 1-94 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 page 1-95 later in this chapter for full details.

See also the chapter entitled *An introduction to SWIs* on page 1-23; and the chapter entitled *Program Environment* on page 1-287 for more about handlers.

UKPLOTV

UKPLOTV is called when a VDU 25,*n* (Plot) or a SWI OS_Plot *n* command is issued with an unknown value of *n*. The default action is to do nothing – unknown VDU 25s (Plots) are usually ignored. See page 1-97 later in this chapter for full details.

MouseV

MouseV is used to indirect all calls to OS_Mouse – see page 1-726 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 (see the chapter entitled *The Font Manager* on page 3-411). If the Font module is disabled, the default action is to do nothing – no output is sent to the VDU. See page 1-98 later in this chapter for full details.

See also the chapter entitled *Character Output* on page 1-503, and the chapter entitled *VDU Drivers* on page 1-547.

TickerV

TickerV is called every centisecond. It must never be intercepted. See page 1-99 later in this chapter for full details.

UpCallV

UpCallV is used to indirect all calls to OS_UpCall – see the chapter entitled *Communications within RISC OS* on page 1-179 for full details. The default action is to call the UpCall handler.

ChangeEnvironmentV

ChangeEnvironmentV is used to indirect all calls to OS_ChangeEnvironment – see page 1-320 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 page 1-788 for full details. The default action is to call the relevant ROM OS_SpriteOp routine. (In fact there are two claimants for this vector: one intercepts those calls handled by the kernel's sprite routines, the another intercepts those handled by the SpriteExtend module.)

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 page 1-100 later in this chapter for full details.

See also the chapter entitled *Draw module* on page 3-533.

EconetV

EconetV is called whenever there is activity on the Econet. The default action is to display the Hourglass on the screen. See page 1-101 later in this chapter for full details.

See also the chapter entitled *Econet* on page 2-619, the chapter entitled *Hourglass* on page 2-745, and the chapter entitled *NetStatus* on page 2-759.

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 page 1-103 later in this chapter for full details.

See also the chapter entitled *ColourTrans* on page 3-333.

PaletteV

PaletteV is called whenever a call is made to read or write the palette. The default action is to call the ROM routine to read or write the palette. See page 1-105 later in this chapter for full details.

This vector has no default owner under RISC OS 2.

SerialV

SerialV is used to indirect all calls to OS_SerialOp – see page 2-468 for full details. The default action is to call the ROM OS_SerialOp routine.

This vector has no default owner under RISC OS 2.

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* on page 1-119 for details of how to add interrupt generating devices to your computer, and the chapter entitled *Program Environment* on page 1-287 for more about handlers.

Related SWIs

None

InsV (Vector &14)

Called to place a byte or block in a buffer

On entry

Byte insertion

R0 = byte to be inserted
R1 = buffer number (bits 0 - 30), with bit 31 clear (⇒ byte operation)

Block insertion

R1 = buffer number (bits 0 - 30), with bit 31 set (⇒ block operation)
R2 = pointer to first byte of data to be inserted
R3 = number of bytes to insert

On exit

Byte insertion

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

Block insertion

R0, R1 preserved
R2 = pointer to remaining data to be inserted
R3 = number of bytes still to be inserted
C flag = 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 byte(s) into a buffer from the system buffers.

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 C flag is used to indicate if the insertion failed; if C=1 then it was not possible to insert all the specified data, or the specified byte.

Block operations are not available in RISC OS 2, nor are they available for buffers that are not handled by the buffer manager.

To use different sized system buffers under RISC OS 2, you must provide handlers for all of InsV, RemV and CnpV. Under later versions of RISC OS you must instead use the buffer manager SWIs Buffer_Create or Buffer_Register.

See also the chapter entitled *Buffers* on page 1-163, and the chapter entitled *The Buffer Manager* on page 4-85.

Related SWIs

OS_Byte 138 (page 1-172), OS_Byte 153 (page 1-178)

RemV (Vector &15)

Called to remove a byte or block from a buffer

On entry

Byte removal

R1 = buffer number (bits 0 - 30), with bit 31 clear (⇒ byte operation)
V flag = 1 if buffer to be examined only, or 0 if data should actually be removed

Block removal

R1 = buffer number (bits 0 - 30), with bit 31 set (⇒ block operation)
R2 = pointer to block to be filled
R3 = number of bytes to place into block
V flag = 1 if buffer to be examined only, or 0 if data should actually be removed

On exit

Byte removal

R0 = next byte to be removed (examine option), or corrupted (remove option)
R1 preserved
R2 = byte removed (remove option), or corrupted (examine option)
C flag = 1 if buffer was empty on entry

Block removal

R0, R1 preserved
R2 = pointer to updated buffer position
R3 = number of bytes still to be removed
C flag = 1 if buffer was empty 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 145 and OS_Byte 152. The default action is to call the ROM routine to examine or remove byte(s) from the system buffers.

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 C flag is used to indicate if the operation failed; if C=1 then it was not possible to remove/examine all the specified data, or the specified byte.

Block operations are not available in RISC OS 2, nor are they available for buffers that are not handled by the buffer manager.

To use different sized system buffers under RISC OS 2, you must provide handlers for all of InsV, RemV and CnpV. Under later versions of RISC OS you must instead use the buffer manager SWIs Buffer_Create or Buffer_Register.

See also the chapter entitled *Buffers* on page 1-163 and the chapter entitled *The Buffer Manager* on page 4-85.

Related SWIs

OS_Byte 145 (page 1-174), OS_Byte 152 (page 1-176)

CnpV (Vector &16)

Called to count the number of entries/amount of space left in a buffer, or to flush the contents of a buffer

On entry

R1 = buffer number
V flag and C flag encode the action:
V flag = 0, C flag = 0 ⇒ return number of entries
V flag = 0, C flag = 1 ⇒ return amount of free space
V flag = 1 ⇒ flush buffer

On exit

R0 corrupted
R1, bits 0 - 7 = least significant 8 bits of count, if V flag = 0 on entry; else preserved
R2, bits 0 - 23 = most significant 24 bits of count, if V flag = 0 on entry; else preserved

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 flush 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	flush 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

This call also copes with buffer manager buffers.

To use different sized system buffers under RISC OS 2, you must provide handlers for all of InsV, RemV and CnpV. Under later versions of RISC OS you must instead use the buffer manager SWIs Buffer_Create or Buffer_Register.

See also the chapter entitled *Buffers* on page 1-163 and the chapter entitled *The Buffer Manager* on page 4-85.

Related SWIs

OS_Byte 15 (page 1-167), OS_Byte 21 (page 1-169), OS_Byte 128 (page 1-170)

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

OS_WriteC (page 1-515)

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 Unused SWI handler, which by default returns a 'No such SWI' error. See the section entitled *Unused SWI* on page 1-295 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 chapter entitled *An introduction to SWIs* on page 1-23; and the chapter entitled *Program Environment* on page 1-287 for more about handlers.

Related SWIs

OS_UnusedSWI (page 1-312)

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 140 - 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

OS_WriteC (page 1-515), OS_ReadVduVariables (page 1-730), OS_Plot (page 1-744)

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 (see the chapter entitled *The Font Manager* on page 3-411). If the Font module is disabled, the default action is to do nothing – no output is sent to the VDU.

See also the chapter entitled *Character Output* on page 1-503, and the chapter entitled *VDU Drivers* on page 1-547.

Related SWIs

OS_WriteC (page 1-515)

TickerV (Vector &1C)

Called every centisecond

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 centisecond. It must never be intercepted, as this would prevent other clients 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 or cope with re-entrancy. One way of ensuring that the code is single threaded 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 dependent on SWI issued
R8 = index of SWI within the Draw module SWI chunk

On exit

Dependent 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* on page 3-533.

Related SWIs

Draw_… (page 3-545 onwards)

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
&70	NetFS_StartBroadcastLoad	R1 = total size of data
&71	NetFS_PartBroadcastLoad	R1 = amount of data transferred
&72	NetFS_FinishBroadcastLoad	
&80	NetFS_StartBroadcastSave	R1 = total size of data
&81	NetFS_PartBroadcastSave	R1 = amount of data transferred
&82	NetFS_FinishBroadcastSave	
&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 chapter entitled *Hourglass* on page 2-745, and the chapter entitled *NetStatus* on page 2-759.

See also the chapter entitled *NetFS* on page 2-343, the chapter entitled *NetPrint* on page 2-393, and the chapter entitled *Econet* on page 2-619.

Related SWIs

Econet_… (page 2-657 onwards), NetFS_… (page 2-350 onwards), NetPrint_… (page 2-397 onwards) and Hourglass_… (page 2-746 onwards)

ColourV
(Vector &22)

Used to indirect all SWI calls made to the ColourTrans module

On entry

R0 - R7 dependent on SWI issued
R8 = index of SWI within the ColourTrans module SWI chunk

On exit

Dependent 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.

The index held in R8 is decoded as follows:

0	ColourTrans_SelectTable
1	ColourTrans_SelectGCOLTable
2	ColourTrans_ReturnGCOL
3	ColourTrans_SetGCOL
4	ColourTrans_ReturnColourNumber
5	ColourTrans_ReturnGCOLForMode
6	ColourTrans_ReturnColourNumberForMode
7	ColourTrans_ReturnOppGCOL
8	ColourTrans_SetOppGCOL
9	ColourTrans_ReturnOppColourNumber
10	ColourTrans_ReturnOppGCOLForMode
11	ColourTrans_ReturnOppColourNumberForMode
12	ColourTrans_GCOLToColourNumber

13	ColourTrans_ColourNumberToGCOL
14	ColourTrans_ReturnFontColours
15	ColourTrans_SetFontColours
16	ColourTrans_InvalidateCache
17	ColourTrans_SetCalibration
18	ColourTrans_ReadCalibration
19	ColourTrans_ConvertDeviceColour
20	ColourTrans_ConvertDevicePalette
21	ColourTrans_ConvertRGBToCIE
22	ColourTrans_ConvertCIEToRGB
23	ColourTrans_WriteCalibrationToFile
24	ColourTrans_ConvertRGBToHSV
25	ColourTrans_ConvertHSVToRGB
26	ColourTrans_ConvertRGBToCMYK
27	ColourTrans_ConvertCMYKToRGB
28	ColourTrans_ReadPalette
29	ColourTrans_WritePalette
30	ColourTrans_SetColour
31	ColourTrans_MiscOp
32	ColourTrans_WriteLoadingsToFile
32	ColourTrans_SetTextColour
32	ColourTrans_SetOppTextColour
32	ColourTrans_GenerateTable

See also the chapter entitled *ColourTrans* on page 3-333.

Related SWIs

ColourTrans_… (page 3-344 onwards)

PaletteV (Vector &23)

Called whenever the palette is to be read or written

On entry

Register usage is dependent on a reason code held in R4:

Read palette

R0 = logical colour
R1 = type of colour (16,17,18,24,25)
R4 = 1 (reason code)

Set palette

R0 = logical colour
R1 = type of colour (16,17,18,24,25)
R2 = 1st flash colour (&BBGGRRxx) – device colour
R3 = 2nd flash colour (&BBGGRRxx) – device colour
R4 = 2 (reason code)

Set first flash state

R4 = 3 (reason code)

Set second flash state

R4 = 4 (reason code)

Set default palette

R4 = 5 (reason code)

On exit

Read palette

R2 = 1st flash colour (&BBGGRRxx) – device colour
R3 = 2nd flash colour (&BBGGRRxx) – device colour
R4 = 0 ⇒ operation complete

Other reason codes

R4 = 0 ⇒ operation complete

Use

This vector is called whenever the palette is to be read or written. Calls this applies to include:

- VDU 19 page 1-588
- OS_Word 12 page 1-706
- OS_ReadPalette page 1-728
- ColourTrans_ReadPalette page 3-393
- ColourTrans_WritePalette page 3-395

By claiming this vector, you can get replacement graphics hardware to intercept such calls, and perform the operation using their own palette. On completion, you should set R4 to zero on exit; RISC OS then knows not to perform the operation itself.

By default, this vector calls the ROM routines to read/write the computer's own palette; they likewise set R4 to zero on exit to notify the caller that the operation was completed.

This vector has no default owner under RISC OS 2 or RISC OS 3 (version 3.00). However, you can write software that calls this vector – and that works correctly under all versions of RISC OS – by checking the value of R4 on exit to see if the operation is complete. If it is not complete, you then need to use your own code to read or write the palette. For more information and example code fragments, see the section entitled *Application notes* on page 1-110.

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 (preprocessing)
- calls the remaining routines on the vector to write the altered character
- toggles the flag (postprocessing)
- 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             ; decide which territory manager table to use
    LDREQ   r1, lowercase_table%
    LDRNE   r1, uppercase_table%
    LDRB    r0, [r1, r0]        ; look up character and put back in r0

; 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 earlie
r
                                ; #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

; reserve space to store the addresses of the territory manager case tables
.lowercase_table%
    EQUD    0
.uppercase_table%
    EQUD    0
]
NEXT
REM Get addresses of the territory manager case tables
SYS “Territory_LowerCaseTable”,-1 TO !lowercase_table%
SYS “Territory_UpperCaseTable”,-1 TO !uppercase_table%
DIM flag% 1
?flag%=0
WrchV%=3
ON ERROR SYS “XOS_Release”, WrchV%, vectorcode%, flag%: PRINTREPORT$: END
SYS “OS_Claim”, WrchV%, vectorcode%, flag%
REPEAT
  INPUT command$
  OSCLI command$
UNTIL command$=””
SYS “XOS_Release”, WrchV%, vectorcode%, flag%
END
```

Application notes

The PaletteV vector has no default owner under RISC OS 2 or RISC OS 3 (version 3.00), but you may still wish to write software that calls this vector, and can hence interact with (say) a replacement graphics card.

The two pieces of code below work correctly under all current versions of RISC OS. They do so by checking the value of R4 on exit from PaletteV to see if the read/write palette operation is complete. If it is not complete, the code is being run on a RISC OS 2 machine, and there was no PaletteV claimant (such as code downloaded from a graphics card) that was able to complete the operation. In such cases, the code then reads/writes the palette itself.

Reading a palette entry

The following piece of code reads a palette entry:

```
; In    R0 = logical colour
;       R1 = type of colour (16,17,18,24,25)
; Out   R2 = 1st flash colour (&BBGGRRxx) - device colour
;       R3 = 2nd flash colour (&BBGGRRxx) - device colour
;       VC => flags preserved, VS => R0->error, flags corrupt
;       (mustn't be called with V set)

readpalette Entry "R4,R9"

        MOV R4,#1               ; read palette
        MOV R9,#PaletteV
        SWI XOS_CallAVector     ; returns &BBGGRRxx
        EXIT VS

        TEQ R4,#0
        EXIT EQ

        SWI XOS_ReadPalette     ; returns &B0G0R0xx

        LDRVC R4,=&F0F0F000     ; clears low nibbles and bottom byte
                                ; (we want to preserve bits 0..7)
        ANDVC R14,R2,R4
        ORRVC R2,R2,R14,LSR #4  ; force to &BBGGRRxx

        ANDVC R14,R3,R4
        ORRVC R3,R3,R14,LSR #4  ; force to &BBGGRRxx

        EXITS VC
        EXIT
        LTORG
```

Note that if the vector is claimed, the resulting colours must be 24-bit, rather than the restricted versions returned by OS_ReadPalette.

Writing a palette entry

The following piece of code writes a palette entry:

```
; In    R0 = logical colour
;       R1 = type of colour (16,17,18,24,25)
;       R2 = 1st flash colour (&BBGGRRxx) - device colour
;       R3 = 2nd flash colour (&BBGGRRxx) - device colour
; Out   VC => flags preserved, VS => R0->error, flags corrupt
;       (mustn't be called with V set)
;
; NB:   Doesn't cope with R1=16,R2<>R3 (write different flash states).
;       It is in fact impossible to get R1=24or25,R2<>R3 to work.

setpalette "R4,R9"

        MOV R4,#2               ; set palette
        MOV R9,#PaletteV
        SWI XOS_CallAVector
        EXIT VS

        TEQ R4,#0
        EXITS EQ

        AND R14,R0,#&FF
        AND R4,R1,#&FF
        ORR R4,R14,R4,LSL #8
        BIC R14,R2,#&FF         ; R14 = &BBGGRR00
        ORR R4,R4,R14,LSL #8    ; R4 = &GGRRr1r0 (green,red,R1,R0)
        MOV R14,R2,LSR #24      ; R14 = &000000BB (blue)
        Push "R0,R1,R4,R14"
        ADD R1,sp,#2*4          ; R1 -> block
        MOV R0,#12              ; write palette
        SWI XOS_Word
        STRVS R0,[sp]
        Pull "R0,R1,R4,R14"

        EXITS VC
        EXIT
```

Note that when writing the palette, there is no need to alter the parameters when calling VDU 19 or OS_Word 12, since these only look at the top nibbles of each gun.

However, 24-bit palette values can only be received through the vector, since the VDU 19 and OS_Word calls cannot trust the values of the bottom nibbles of the palette values passed to them, and must treat them as being copies of the corresponding top nibbles.

9 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. When the exception occurs, before any of these vectors is called, the ARM processor saves the current program counter (R15) to R14_svc. The ARM is then forced to SVC mode, and interrupts are disabled.

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 exception dump area, 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.

These vectors may be set and read as described in the chapter entitled *Program Environment* on page 1-287. Very few programs need to take account of them.

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. 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.

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.

Address exception vector

The address exception vector is called when a data reference is made outside the range 0 - &3FFFFFF.

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 processor saves the current program counter (R15) to R14_svc. The ARM is then 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* on page 1-23.

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 processor saves the current program counter (R15) to R14_irq, the ARM is forced to IRQ mode, and interrupts are disabled.

For full details, see the chapter entitled *Interrupts and handling them* on page 1-119.

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 processor saves the current program counter (R15) to R14_fiq, the ARM is forced to FIQ mode, and both normal and fast interrupts are disabled.

For full details, see again 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 an

```
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	

10 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. One is 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 RISC OS computers 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 correspond directly to bits of the interrupt registers held in the IOC chip:

Device number	Corresponds to:
0	Bit 0 of IRQ A registers
1	Bit 1 of IRQ A registers
…	
7	Bit 7 of IRQ A registers
8	Bit 0 of IRQ B registers
9	Bit 1 of IRQ B registers
…	
15	Bit 7 of IRQ B registers

See the section entitled *IOC registers* on page 1-143 for more details.

Not all RISC OS computers use the same interrupt generating hardware. Early models (eg the Archimedes 300, 400 and 500 series, and the A3000) use a variety of peripheral control chips; current models (eg the A5000) use the 82C710 or 82C711 chip; future models may use other peripheral controllers. These different peripheral controllers are mapped differently onto the IOC chip's IRQ registers. Consequently, device numbers differ between models of RISC OS computers.

You can distinguish between the different peripheral controllers and their properties by calling OS_ReadSysInfo (page 1-746) using its various reason codes.

For early models (ie the Archimedes 300, 400 and 500 series, and the A3000), the device numbers are:

0	Printer Busy
1	Serial port Ringing Indicator
2	Printer Acknowledge
3	VSync Pulse
4	Power on reset – this should never appear in normal use
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 interrupt
11	Hard disc controller interrupt
12	Floppy disc changed
13	Expansion card interrupt
14	Keyboard serial transmit register empty
15	Keyboard serial receive register full

For models using the 82C710 or 82C711 peripheral controller (eg the A5000), the device numbers are:

0	Printer interrupt from 82C710/711
1	Low battery warning
2	Floppy disc Index
3	VSync Pulse
4	Power on reset – this should never appear in normal use
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 interrupt from 82C710/711 – also mapped to FIQ device 4
11	IDE hard disc interrupt
12	Floppy disc interrupt from 82C710/711
13	Expansion card interrupt
14	Keyboard serial transmit register empty
15	Keyboard serial receive register full

(Device numbers 3 - 9 and 13 - 15 have the same meaning as for early models.)

Note that RISC OS 2 does not support the 82C710 or 82C711 peripheral controller.

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 (ie one that uses the same code and workspace).

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.

IRQUtils

After the release of RISC OS 2, a module called IRQUtils was released to improve interrupt latency.

The changes this made have now been incorporated in RISC OS ’s kernel. A dummy module named IRQUtils has been included with an appropriate version number, so that applications written to run under RISC OS 2 that check for its presence will still run correctly.

SWI Calls

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 (ie an expansion card)
R4 = interrupt mask to use, if R0 = 8 or 13 on entry (ie an expansion card)

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 page 4-127 onwards of the chapter entitled *Expansion Cards and Extension ROMs*. 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 (page 1-125)

Related vectors

None

OS_ReleaseDeviceVector (SWI &4C)

Releases a device vector

On entry

R0 = device number
R1 = address of device driver routine
R2 = R12 value
R3 = interrupt location if R0 = 8 or 13 on entry (ie an expansion card)
R4 = interrupt mask if R0 = 8 or 13 on entry (ie an expansion card)

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 the list of claimants of a device vector. The device driver is identified by the contents of the registers on entry; R0 - R2 (R0 - R4 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 within IOC when you release its vector. For expansion card IRQs (ie R0 = 8 or 13 on entry), you should prevent your device from interrupting again by programming the hardware on your expansion card.

Related SWIs

OS_ClaimDeviceVector (page 1-123)

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 intercepted 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 in the section entitled *IOC registers* on page 1-143.

Device driver routines

Entry conditions

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.

Servicing the interrupt and returning

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 cannot simply do `TEQP PC,#I_bit`, because in this instruction the PC is presented without the PSR bits. For more details see *Appendix A: ARM assembler* on page 4-363.

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 (ie re-entrancy occurring).

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. A 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
MOV     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
MOV     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. If you want to check flags on exit from the SWI (eg the V flag), you must do so before you re-enter the original processor mode, as this method restores not just the mode bits, but also all the original flags.

An alternative method is shown below. It is one instruction longer, but only uses one temporary register (R8), and allows testing the flags returned by the SWI after restoring the original mode:

```
MOV     R8, PC                  ; R8 holds PC and PSR
AND     R8, R8, #3              ; Extract current mode bits
EOR     R8, R8, #SVC_Mode       ; R8 = current mode EOR SVC_Mode
TEQP    R8, PC                  ; Enter SVC mode
MOV     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    R8, PC                  ; Restore mode, preserving other flags
MOV     R0, R0                  ; No-op to prevent contention
```

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 of re-entrancy

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 conditions

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 *interrupt request*, or 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 interrupts (ie 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

Devices handled under FIQs also have device numbers. Again, the device numbers correspond to the bits in IOC registers: these are the FIQ interrupt registers.

Device number	Corresponds to:
0	Bit 0 of FIQ registers
1	Bit1 of FIQ registers
…	
7	Bit 7 of FIQ registers

Just like IRQ device numbers, FIQ device numbers differ between models of RISC OS computers, depending on the peripheral controller chips used. For early models (eg the Archimedes 300, 400 and 500 series, and the A3000), the FIQ device numbers are:

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

For models using the 82C710 or 82C711 peripheral controller (eg the A5000), the FIQ device numbers are:

0	Floppy DMA data request
1	FH1 pin on IOC
2	Econet interrupt
3	C3 pin on IOC
4	Serial port interrupt from 82C710/711 – also mapped to IRQ device 10
5	C5 pin on IOC
6	Expansion card interrupt
7	Force FIQ – this bit is always set, but usually masked out

(FIQ device numbers 2, 3 and 5 - 7 have the same meaning as for early models.)

Again we make the point that RISC OS 2 does not support the 82C710 or 82C711 peripheral controller.

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.

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 the IOC's fast interrupt status register is set.

3 The IOC's fast interrupt status register is ANDed with its fast interrupt mask register, and the result put in its request register.

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 (ie IRQs). 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 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
MOV     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.

Service Calls

Service_ReleaseFIQ (Service Call &0B)

Release FIQ

On entry

R1 = &0B (reason code)

On exit

R1 = 0 to claim, else preserved to pass on

Use

This service call must be issued by any module immediately after it releases the FIQ hardware vector. You may claim this service call if you wish to usurp the default FIQ owner (the Econet module) and install your own code on the FIQ hardware vector.

If no module claims this service call, then Econet does so, and installs its own code on the FIQ hardware vector. Should even Econet not claim the service call – for example if the Econet module has been unplugged – then the kernel installs its default FIQ handler.

See the section entitled *Using FIQs* on page 1-133 for details of other steps to take when claiming or releasing the FIQ hardware vector, and also the chapter entitled *Hardware vectors* on page 1-113 for additional information about the vector.

Service_ClaimFIQ (Service Call &0C)

Claim FIQ

On entry

R1 = &0C (reason code)

On exit

R1 = 0 to claim, else preserved to pass on

Use

This service call must be issued by any module running as a foreground task (ie not as an IRQ process) that wishes to claim the FIQ hardware vector.

It informs the current FIQ owner that it must release the vector as soon as it can cleanly do so. The current owner must complete without disruption any unfinished FIQ processing, release the vector, and then claim the service call by setting R1 to zero. As soon as the claimant finds that the service call has been claimed, it knows it has claimed the FIQ hardware vector.

See the section entitled *Using FIQs* on page 1-133 for details of other steps to take when claiming or releasing the FIQ hardware vector, and also the chapter entitled *Hardware vectors* on page 1-113 for additional information about the vector.

Service_ClaimFIQinBackground (Service Call &47)

Claim FIQ in background

On entry

R1 = &47 (reason code)

On exit

R1 = 0 to claim, else preserved to pass on

Use

This service call must be issued by any module running as a background task (ie as an IRQ process) that wishes to claim the FIQ hardware vector. It may also be issued by foreground tasks that wish to poll the FIQ vector for availability. Unlike Service_ClaimFIQ, this call may return with R1 preserved (ie not claimed), meaning that the current FIQ owner has not released the vector.

The service call informs the current FIQ owner that it must release the vector if it can immediately do so. If the current owner is busy with a FIQ, it must take no action, merely passing on the service call; if however it is idle, it may release the vector and then claim the service call by setting R1 to zero. If the claimant finds that the service call has been claimed, it knows it has successfully claimed the FIQ hardware vector; however, if the claimant finds that the service call has not been claimed, it knows the current owner has not released the FIQ hardware vector, in which case the claimant may reissue the service call at a later time.

Background claims are released by Service_ReleaseFIQ, as before.

See the section entitled *Using FIQs* on page 1-133 for details of other steps to take when claiming or releasing the FIQ hardware vector, and also the chapter entitled *Hardware vectors* on page 1-113 for additional information about the vector.

SWI Calls

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 (page 1-140)

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 (page 1-139)

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. This leaves you using the SVC stack. 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* on page 1-9
- consult the *Acorn RISC Machine family Data Manual*. VLSI Technology Inc. (1990) Prentice-Hall, Englewood Cliffs, NJ, USA: ISBN 0-13-781618-9.
- consult the datasheets for the various peripheral controllers used, available from their manufacturers
- contact Acorn Customer Service.

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

Figure 10.1 Typical memory mapping of IOC registers

Control register

The IOC chip's control register allows you to read and write its six external control pins C0 - C6, and to read two other pins. Again there are differences between models of RISC OS computers, depending on the peripheral controllers used. For early models (eg the Archimedes 300, 400 and 500 series, and the A3000), the bits of the control register are mapped as follows:

Bit	Function
0	IIC serial bus data
1	IIC serial bus clock
2	Floppy disc ready
3	Reset enable (A540/R200 series only)
4	Current level of C4 pin on IOC (available on Auxiliary I/O connector)
5	Speaker mute
6	Current level of –IF pin on IOC (Printer Acknowledge signal)
7	Current level of IR pin on IOC (Vertical Flyback signal)

For models using the 82C710 or 82C711 peripheral controller (eg the A5000), the bits of the control register are mapped as follows:

Bit	Function
0	IIC serial bus data
1	IIC serial bus clock
2	Floppy disc density
3	Unique machine ID chip – control pin
4	Serial FIQ
5	Speaker mute
6	Current level of –IF pin on IOC (Floppy disc Index signal)
7	Current level of IR pin on IOC (Vertical Flyback signal)

Again we make the point that RISC OS 2 does not support the 82C710 or 82C711 peripheral controller.

Other devices

Other devices and peripheral controllers are mapped into memory in these locations on early model RISC OS computers (eg the Archimedes 300, 400 and 500 series, and the A3000):

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 B
&3350040	Fast	5	HC574	Latch A
&33A0000	Sync	2	6854	Econet controller
&33B0000	Sync	3	6551	Serial port controller
&33C0000	Sync	4	—	Internal Expansion cards

Figure 10.2 Early memory mapping of non-IOC devices and peripheral controllers

Current models that use the 82C710 or 82C711 (eg the A5000) use this mapping:

Address	Type	Bank	IC	Use
&3010000			82C710/1	Peripheral controller
&3012000			82C710/1	Floppy disc DMA control
&3240000	Slow	4	—	Internal Expansion cards
&3270000	Slow	7	—	External Expansion cards
&32C0000	Med	4	—	Internal Expansion cards
&3340000	Fast	4	—	Internal Expansion cards
&3350048	Fast	5	—	Video clock/[Sync polarity]
&3350050	Fast	5	—	ASIC presence
&3350054	Fast	5	—	[Clock speed]
&3350070	Fast	5	—	Monitor ID field
&3350074	Fast	5	—	VGA test pin/SCART sound
&33A0000	Sync	2	6854	Econet controller
&33C0000	Sync	4	—	Internal Expansion cards

Figure 10.3 Typical memory mapping of non-IOC devices and peripheral controllers

11 Events

Introduction

Events are used by RISC OS to indicate that something specific has occurred. These are typically generated using the SWI OS_GenerateEvent when RISC OS services an interrupt. 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
18	Reserved for use by an external developer
19	Internet has generated an event
20	Reserved for use by an external developer
21	Reserved for use by an external developer
22	Device overrun
23	Reserved for use by an external developer
24	A driver has received a frame for the Internet module
25	A driver has completed an Internet transmission request
26	Reserved for use by Acorn
27	Reserved for use by Acorn
28	Portable BMU has received 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* on page 1-63 for further details of vectors. 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 they start processing another. The use of certain operating system calls must be avoided. For further details see the section entitled *Restrictions* on page 1-128.

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.

SWI Calls

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 (page 1-152), OS_GenerateEvent (page 1-154)

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 (page 1-150), OS_GenerateEvent (page 1-154)

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 (page 1-150), OS_Byte 14 (page 1-152)

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 (e.g. printer buffer, serial port output buffer) which has output empty events enabled, or an attempt is made to remove another character from the buffer once it has been emptied. See the chapter entitled *Buffers* on page 1-163.

Input buffer full event

R0 = 1
R1 = buffer number (bits 0 - 30) and byte/block operation flag (bit 31):
bit 31 clear ⇒ byte operation – R2 holds byte
bit 31 set ⇒ block operation – R2 points to block of length R3
R2 = byte that could not be inserted into buffer (if byte operation);
else R2 = pointer to data not inserted (if block operation)
R3 = number of bytes not inserted (if block operation)

This event is generated when an input buffer (which has input full events enabled) is full and when the operating system tries to enter a character into the buffer but fails. See the chapter entitled *Buffers* on page 1-163.

Block operations do not occur in RISC OS 2, nor do they occur for buffers that are not handled by the buffer manager.

Character input event

R0 = 2
R1 = buffer number (0 for keyboard, or 1 for serial input)
R2 = byte being inserted into buffer

This event is generated when OS_Byte 153 is called to insert any character except Escape into the keyboard buffer, or into the serial buffer when serial input is treated as keyboard input (ie OS_Byte 181 is set to zero). If the character is Escape, then the Escape event is instead generated.

See the chapter entitled *Character Input* on page 1-863 for a description of buffer values for the keyboard buffer.

OS_Byte 153 is called by the keyboard driver to insert keys into the keyboard buffer, and by the serial device driver when it receives a character.

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 (SWI &07)* on page 1-423 for details of the interval timer.

The interval timer is obsolescent, since there is only one provided, which is not very useful in a multi-tasking environment. You should instead use OS_CallAfter (page 1-441) or OS_CallEvery (page 1-443).

Escape event

R0 = 6

This event is generated either when Esc is pressed, or when an escape condition is received from the RS423 input port and OS_Byte 181 is set to zero. See the chapter entitled *Character Input* on page 1-863 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 chapter entitled *Econet* on page 2-619 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 a low-level internal key number transmitted by the keyboard to the IOC device, and 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:

high		0	1	2	3	4	5	6	7
low	0	Esc	‘	Home	P	G	C	Alt (R)	Select
	1	F1	1	PageUp	[	H	V	Ctrl (R)	Menu
	2	F2	2	NumLock	]	J	B	←	Adjust
	3	F3	3	/	\	K	N	↓	
	4	F4	4	*	Delete	L	M	→	
	5	F5	5	#	Copy	:	,	0	
	6	F6	6	Tab	PageDown	"	.	.	
	7	F7	7	Q	7	Return	/	Enter	
	8	F8	8	W	8	4	Shift (R)		
	9	F9	9	E	9	5	↑		
	A	F10	0	R	-	6	1		
	B	F11	-	T	Ctrl (L)	+	2		
	C	F12	=	Y	A	Shift (L)	3		
	D	Print	£	U	S		CapsLock		
	E	ScrollLock	↤	I	D	Z	Alt (L)		
	F	Break	Insert	O	F	X	Space		

Figure 11.1 Low-level internal key numbers

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. 4D is unused on the UK model, but may be used on some other models for an extra key.

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 entitled *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* on page 4-3 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* on page 2-619 for further 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* on page 2-619 for further 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* on page 2-619 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 about 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.

Internet event

R0 = 19
R1 = event code
R2 = socket descriptor

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

0	A socket has input waiting to be read
1	An urgent event has occurred, such as the arrival of out-of-band data
2	A socket connection is broken.

These events only occur if you are using the Internet module supplied with the *TCP/IP Protocol Suite*. See the *TCP/IP Programmer's Guide* for further details.

Device overrun event

R0 = 22
R1 = device driver's handle
R2 = file handle
R3 = 0

This event is generated when the SWI DeviceFS_ReceivedCharacter is called on an unbuffered stream, and the previously received character has not been read. The new character overwrites any previous character.

Internet receive event

R0 = 24
R1 = pointer to data buffer chain containing Rx data
R2 = pointer to name of interface controlled by driver ('et', 'en', etc)
R3 = physical unit number (0 - 3)
R4 = Rx frame type

This event is generated by a driver to indicate to the Internet module that it has received and stored a frame from the network.

Internet transmission status event

R0 = 25
R1 = pointer to data buffer chain containing Tx data
R2 = pointer to name of interface controlled by driver ('et', 'en', etc)
R3 = physical unit number (0 - 3)
R4 = error number (driver specific), or 0 if successful

This event is generated by a driver to indicate to the Internet module that it has completed a transmission request. It does not necessarily imply successful transmission and reception by the target host.

Portable BMU interrupt event

R0 = 28
R1 = status bits for BMU variable 10 (see page 4-213)

This event is generated by the Portable module when it receives interrupts from the BMU (battery management unit). You must not claim it yourself. See the chapter entitled *The Portable module* on page 4-205 for further details.

12 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.

The buffer manager

The buffer manager is a global buffer managing system used by DeviceFS to provide buffers for the various devices that can be accessed. It provides a set of calls for setting up a buffer, inserting and removing data from a buffer, and removing a buffer. For more details about the buffer manager see the chapter entitled *The Buffer Manager* on page 4-85.

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* on page 1-863.

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

† From RISC OS 3 onwards, the size of the printer buffer is configurable using *Configure PrinterBufferSize.

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 by RISC OS. 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, although top-bit-set characters are treated specially in the keyboard buffer (and optionally in the serial input buffer). See the chapter entitled *Character Input* on page 1-863 for a description of buffer values for the keyboard buffer. The mouse buffer contents refer to buffered button 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 – see page 1-174.

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 on page 1-88 onwards of 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.

Changing buffer sizes

To use different sized system buffers under RISC OS 2, you must provide handlers for all of InsV, RemV and CnpV. You could do so in a module that 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.

Under later versions of RISC OS you can alter the size of the printer buffer using *Configure PrinterBufferSize. To alter the size of other system buffers, rather than claiming InsV, RemV and CnpV, you must instead use the buffer manager SWIs Buffer_Create or Buffer_Register.

SWI calls

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 (page 1-169)

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 (page 1-167)

Related vectors

ByteV

OS_Byte 128 (SWI &06)

Gets mouse coordinates, or number of bytes in an input buffer, or number of free bytes in an output buffer

On entry

R0 = 128
R1 = reason code

On exit

R0 preserved
R1, bits 0 - 7 = low 8 bits of answer
R2, bits 0 - 23 = high 24 bits of answer

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

On entry	**On exit R1 & R2 contain the:**
R1 = 7	mouse x position
R1 = 8	mouse y position
R1 = 246	number of bytes in the mouse buffer
R1 = 252	number of bytes free in the printer buffer

R1 = 253	number of bytes free in the RS423 output buffer
R1 = 254	number of bytes in the RS423 input buffer
R1 = 255	number of bytes in the keyboard buffer

Obviously we are more concerned with the calls where R1 ≥ 246 here. Note that R1 = (255 – buffer number) in these cases. If you want, you can also calculate this as { (– (buffer number + 1)) AND &FF }.

The calls to read the mouse position are considered obsolete, and are unreliable because they read the buffered position rather than the actual position. You should use OS_Mouse (page 1-726) instead.

Related SWIs

None

Related vectors

ByteV, CnpV

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

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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 reduce latency in the interrupt system.

Related SWIs

OS_Byte 145 (page 1-174), OS_Byte 152 (page 1-176), OS_Byte 153 (page 1-178)

Related vectors

ByteV, InsV

OS_Byte 145 (SWI &06)

Gets a byte from a buffer

On entry

R0 = 145
R1 = buffer number

On exit

R0, R1 preserved
R2 = byte extracted
C flag = 0 if byte read
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 extracts the next byte from a specified buffer. If the buffer was empty then the C flag is set, and R2 will be invalid.

This call assumes the buffer number is correct; it does not generate an error if passed a bad buffer number, and its behaviour is undefined.

Related SWIs

OS_Byte 138 (page 1-172), OS_Byte 152 (page 1-176), OS_Byte 153 (page 1-178)

Related vectors

ByteV, RemV

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.

This call assumes the buffer number is correct; it does not generate an error if passed a bad buffer number, and its behaviour is undefined.

Related SWIs

OS_Byte 138 (page 1-172), OS_Byte 145 (page 1-174), OS_Byte 153 (page 1-178)

Related vectors

ByteV, RemV

OS_Byte 153 (SWI &06)

Inserts a byte into one of the two input buffers

On entry

R0 = 153
R1 = buffer number (0 or 1)
R2 = byte to insert

On exit

R0 preserved
R1, R2 corrupted
C flag = 0 if byte inserted
C flag = 1 if buffer was full

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

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

If the buffer was full and a byte could not be inserted, then the C flag is set on return.

If the current escape character (usually ASCII 27) is inserted, then appropriate action is taken; see the chapter entitled *Character Input* on page 1-863.

Related SWIs

OS_Byte 138 (page 1-172), OS_Byte 145 (page 1-174), OS_Byte 152 (page 1-176)

Related vectors

ByteV, InsV

13 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.

Vectors

OS_CallAVector is used to call the routine(s) on a software vector. This SWI and the calls necessary to add routines to a vector have already been described in the chapter entitled *Software vectors* on page 1-63.

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 some examples of OS_ServiceCall – the reason codes to claim and release FIQs.

This call is fully documented on page 1-254 onwards of 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_SendMessage).

These calls are all fully documented in the chapter entitled *The Window Manager* on page 3-3.

Call Backs

CallBacks are routines that are called when RISC OS is threaded out. There are two types:

- Transient CallBacks are set up using OS_AddCallBack (page 1-324). They are called once only, and may be called when RISC OS is threaded but idle. They might be used by an interrupt handling routine that is unable to do some things itself (eg calling a re-entrant SWI, or performing a long operation that would unacceptably increase interrupt latency), but wishes to have a routine ‘called back’ later to do these things.
- Non-transient CallBacks are set up using OS_SetCallBack (page 1-313). They are handled by the CallBack handler; you can replace the default one using OS_ChangeEnvironment (page 1-320).

For full details, see the chapter entitled *Program Environment* on page 1-287.

UpCalls

The kernel provides the SWI OS_UpCall, which warns applications of particular situations. It is described below.

OS_UpCall (SWI &33)

Calls that RISC OS makes up to an application to warn of particular situations

On entry

R0 = reason code
Other registers are reason code dependent

On exit

R0 preserved
Other registers are reason code dependent

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 SWI 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* on page 1-63), or install an UpCall handler (see the chapter entitled *Program Environment* on page 1-287).

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 generally occur in the foreground, and are hence different to Events, which occur in the background.

The reason code in R0 may be one of the following:

Code	Meaning	Page
1	Media not present (ie previously used but no longer accessible)	1-183
2	Media not known (ie not previously used)	1-183
3	File is being modified	1-185
4	Media search end (ie medium supplied, or operation cancelled)	1-190
6	Task wants to sleep until some termination condition is met	1-191
7	Open pipe has been closed or deleted	1-192
8	Buffer filling (ie free space has become less than threshold)	1-193
9	Buffer emptying (ie free space has become more than threshold)	1-194
10	Stream created	1-195
11	Stream closed	1-196
256	Application is starting	1-197
257	RISC OS would like to move memory	1-198

For full details of each reason code, see the entries on the given pages.

Some of the above 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.

Related SWIs

None

Related vectors

UpCallV

OS_UpCall 1 and 2 (SWI &33)

Warns your program that a filing medium 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 (see page 2-21)
R2 = pointer to a null-terminated medium 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 medium type string

On exit

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

Use

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

- a filing medium that it has previously used but can no longer access (R0 = 1)
- a filing medium 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 medium with a string built up using:
  - **1** the medium type string (passed in R6)
  - **2** the filing system name (obtained by calling XOS_FSControl 33 acting on the value of R1 – see page 2-118 for details)
  - **3** the medium 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 medium, 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 medium

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 medium was not found.
- If R0 = 0, then the routine checks for you that the medium 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 medium 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 medium 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 (page 1-190)

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

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
258	Opening for update
259	Closing file
512	Ensuring file's size
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 - 8 are caused by calls to OS_File (page 2-32)
- reason codes 257 - 259 are caused by calls to OS_Find (page 2-75)
- reason codes 512 - 521 are caused by calls to OS_FSControl (page 2-80).

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
- object names will also be canonicalised, except under RISC OS 2.

This UpCall is made **before** the operation, which may subsequently fail. For example, you may receive a rename UpCall for a locked file, which will subsequently fail to rename (because it's locked). 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
R2 = load address
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)
R4 = number of entries (0 for default)
R6 = pointer to special field (or 0)

R9 = 257

Creating and opening for update

R1 = pointer to filename
R2 = external handle that file will be given (if successfully opened)
R6 = pointer to special field (or 0)

R9 = 258

Opening for update

R1 = pointer to filename
R2 = external handle that file will be given (if successfully opened)
R6 = pointer to special field (or 0)

R9 = 259

Closing file

R1 = external handle

R9 = 512

Ensuring file's size

R1 = external handle
R2 = size to ensure
R8 = filing system information word

R9 = 520

Renaming file

R1 = pointer to current object name
R2 = pointer to desired object name
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)
R6 = pointer to special field (or 0)

Related SWIs

None

OS_UpCall 4 (SWI &33)

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

On entry

R0 = 4 (Media search end)

On exit

R0 preserved

Use

This call is made by RISC OS to inform your program that a missing filing medium 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 (page 1-183)

OS_UpCall 6 (SWI &33)

Informs the TaskWindow module that a task wants to sleep until some termination condition is met

On entry

R0 = 6 (Sleep)
R1 = pointer to poll word (in a global memory area, eg the RMA)

On exit

R0 = 0 if UpCall claimed

Use

This call is made by a task that wants to sleep until some termination condition is met, signalled by the contents of the poll word becoming non-zero. It is not available in RISC OS 2.

Control **may return** to the task before the poll word becomes non-zero, but is only **guaranteed to return** if and when the poll word becomes non-zero.

While the task is sleeping other tasks will continue to be polled by the Wimp.

If the termination condition can be recognised externally (ie in another Wimp task or under interrupt) hence causing the poll word to be set non-zero, the calling task should set the poll word to zero on entry. Otherwise the poll word must be non-zero on entry, so that control will return to the calling task after each Wimp Poll.

Note that a task must not use this UpCall if it is not re-entrant, or may have been called by a task which is not re-entrant.

The calling task must be running in a task window. The TaskWindow module intercepts this UpCall; you should not do so yourself. These two restrictions may be removed in future versions of RISC OS.

Related SWIs

OS_UpCall 7 (page 1-192)

OS_UpCall 7 (SWI &33)

Informs the TaskWindow module that an open pipe has been closed or deleted

On entry

R0 = 7 (Sleep no more)
R1 = pointer to poll word (in a global memory area, eg the RMA)

On exit

R0 preserved if V flag clear
R0 = pointer to error block if V flag set

Use

This call is made by PipeFS if an open pipe is closed or deleted. It is not available in RISC OS 2.

The TaskWindow module then traps this and objects if any of its tasks are currently waiting for the poll word related to that pipe to become non-zero, by returning an error.

This prevents a *Shut command from deleting the workspace which is being accessed by the TaskWindow, which could potentially cause address exceptions.

Related SWIs

OS_UpCall 6 (page 1-191)

OS_UpCall 8 (SWI &33)

A buffer's free space has become less than its specified threshold

On entry

R0 = 8 (Buffer filling)
R1 = buffer handle
R2 = 0

On exit

All registers preserved

Use

The Buffer Manager issues this call when data is inserted into the specified buffer, and the free space becomes less than its current threshold. For full details of buffer handles and thresholds, see the chapter entitled *The Buffer Manager* on page 4-85.

This call is never issued under RISC OS 2.

Related SWIs

OS_UpCall 9 (page 1-194)

OS_UpCall 9 (SWI &33)

A buffer's free space has become greater than or equal to its specified threshold

On entry

R0 = 9 (Buffer emptying)
R1 = buffer handle
R2 = –1

On exit

All registers preserved

Use

The Buffer Manager issues this call when data is removed from the specified buffer, and the free space becomes greater than or equal to its current threshold. For full details of buffer handles and thresholds, see the chapter entitled *The Buffer Manager* on page 4-85.

This call is never issued under RISC OS 2.

Related SWIs

OS_UpCall 8 (page 1-193)

OS_UpCall 10 (SWI &33)

Stream created

On entry

R0 = 10 (Stream created)
R1 = device driver's handle
R2 = 0 if created for reception (else created for transmission)
R3 = file handle for stream
R4 = DeviceFS stream handle, as passed to device driver on initialisation

On exit

All registers preserved

Use

DeviceFS issues this call when a stream is created. It serves as a broadcast, and all registers should be preserved. For full details of device handles and streams, see the chapter entitled *DeviceFS* on page 2-429.

This call is never issued under RISC OS 2.

Related SWIs

OS_UpCall 11 (page 1-196)

OS_UpCall 11 (SWI &33)

Stream closed

On entry

R0 = 11 (Stream closed)
R1 = device driver's handle
R2 = 0 if closed for reception (else closed for transmission)
R3 = file handle for stream
R4 = DeviceFS stream handle, as passed to device driver on initialisation

On exit

All registers preserved

Use

DeviceFS issues this call when a stream is closed. It serves as a broadcast, and all registers should be preserved. For full details of device handles and streams, see the chapter entitled *DeviceFS* on page 2-429.

This call is never issued under RISC OS 2.

Related SWIs

OS_UpCall 10 (page 1-195)

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

Use

This call is made just before a new application is going to be started in the current application space – 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 (see the chapter entitled *Software vectors* on page 1-63), or install an UpCall handler (see the chapter entitled *Program Environment* on page 1-287).

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. Again, see the chapter entitled *Program Environment* on page 1-287.

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

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

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 (see the chapter entitled *Software vectors* on page 1-63), or install an UpCall handler (see the chapter entitled *Program Environment* on page 1-287). 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

OS_ChangeDynamicArea (page 1-384)

Part 2 – The kernel

14 Modules

Introduction

A relocatable module is a piece of software which, when loaded into the machine acts as either an extension to the operating system or a replacement to an existing module in the operating system. Modules can contain programming languages or filing systems; they can be used to add new SWIs and * Commands.

Relocatable modules run in an area of memory known as the Relocatable Module Area (RMA) which is maintained by RISC OS. They are ‘relocatable’ because they can be loaded at any particular location in memory. Their code must therefore be relocatable.

RISC OS provides facilities for integrating modules in such a way that, to the user, they appear to be a full part of the system. For instance, the operating system responds to the *Help command, extracting automatically any relevant help text.

Several SWIs and * Commands are provided by the operating system for handling modules, such as one to load a module file from the filing system.

A major piece of software written for RISC OS should only be designed as a module if it fulfils the following requirements:

- it is an extension to RISC OS or an enhancement to an existing RISC OS module
- it is shared by many applications; for example the shared C library
- it needs to be persistently RAM resident over many invocations (even then you should try to do this another way)
- it is small enough

or if:

- it is a desktop application – or part of one – which cannot be paged out (eg it has Econet control blocks active).

  Such programs must use RMA for workspace, and are hence easiest to write as modules.

This chapter describes what is needed to write a module.

Overview

This chapter is divided into two basic areas; using modules and writing them.

Using modules

Use of modules is centralised around the SWI OS_Module. This contains a number of operations that can:

- load, initialise, run and remove a module
- examine and change the amount of RMA space used by a module
- examine module details
- modify instantiations of modules.

All of the operations that a program is likely to need to operate with modules are in this SWI. You could treat the RMA as a kind of filing system, since there are commands to load things into it, remove them and run them.

Some modules are supplied with the computer in ROM. These may be ‘unplugged’ and upgraded versions of them loaded into RMA. They may also be deliberately copied from ROM into RMA, since modules in RAM will execute significantly quicker than in ROM.

There are a number of * Commands that replicate several OS_Module commands at a command line level. You can also obtain convenient lists of all modules currently in the RMA and the system ROM using a * Command.

Instantiation

A module may be initialised more than once. This means that whilst only a single copy of the code is kept in memory, multiple copies of its workspace are created. The workspace is the area where all the data used by the module for dynamic storage is kept. Note that constant data, such as lookup tables is kept inside the main body of the module, with the code. Changing which workspace is used changes the context of the module and allows it to be used for several purposes concurrently. Each copy of the workspace, coupled with the code, is referred to as an instantiation. A module is deemed to be reincarnated when a new instantiation is created.

Only a single copy of the code is needed because it is not changed by being used concurrently. The data is the only thing that provides the context for an initialised module.

An example of the use of instantiations is in the module FileCore. This module provides a core of commands that are common to all filing systems with an ADFS structure, ie ADFS and RAMFS. It appears in one instantiation for each filing system that is using it.

For example, typing *Modules, you can see all the modules that are currently loaded, including the various instantiations of the FileCore module:

```
*Modules
...
28 03839698  018114C4  FileCore%RAM
   03839698  01804374  FileCore%ADFS
   03839698  00000000  FileCore%Base
...
```

This enables you to refer to particular instantiations of a module. For example:

```
*RMKill WaveSynth%Base
```


Writing a module

The core of all modules is the module header. It is a table of 11 entries, each a word in length. These are called by RISC OS to communicate with the module.

Module header

The entries in the header table describe the following things in the module. All but one are offsets to code or some larger piece of data, such as a string, or table:

- Where to start executing in the module. This is used by languages and applications.
- Where to call initialisation code. This has to be called before all the others.
- Where to call finalisation code. This is called before removing the module. It allows the module to shutdown any hardware it is using and generally tidy up.
- A title for the module.
- A help string. This is used automatically by RISC OS when help is requested.
- Detailed help on * Commands.
- Entry points for * Commands. RISC OS will decode the * Commands and call the right entry point for a command for you.
- A table to convert to and from SWI names and numbers.
- Entry points for all the SWIs in the module.
- The chunk number for the module. This is the number that is the base for SWI numbers. There can be up to 64 SWIs in a module, all offsets from this chunk number. This is the only entry in the header that isn't an offset.
- Service call entry (see below).

All communication from RISC OS to a module takes place through this table. As you can see, several features are used by RISC OS without you having to write code to deal with them, such as the help text, and SWI names to numbers conversion.

Service calls

A number of special occurrences in RISC OS are passed around all the modules by RISC OS. Some of these can be claimed. This means that if a module decides that it wants to take control of that occurrence then it stops it being passed on to the rest of the modules. Others cannot be claimed and are used by RISC OS to broadcast some occurrence to all modules. Here is a brief list of the kinds of things that can be sent as service calls. The first part are claimable service calls:

- Unknown command, OS_Byte, OS_Word, *Configure or *Status.
- *Help has been called. This allows you to replace this command when you detect a particular help call being made.
- Memory controller about to be remapped. This allows an application to stop a memory remapping if it doesn't want it to happen.
- Application is about to start. This allows a module to prevent an application from starting. With this, a module could prevent any other tasks running.
- Lookup file type. This converts the 3 byte file type into a string, such as 'BASIC' or 'Text'.
- Various international services, such as handling different alphabets and keyboards.
- The fast interrupt handler has been claimed/released. This is used by device drivers for high data rate devices that depend on the state of the fast interrupt system.

These are the service calls that cannot be claimed and are used to allow modules to perform some action to cope with the occurrence, without stopping it being passed on to all modules:

- An error has occurred. This is called before the error handler, but is only for module's information, not claiming.
- Reset is about to happen/has just happened.
- Filing system re-initialise. This is called when FileSwitch has been re-initialised and this is broadcast to all filing systems that use it to do the same. This is necessary, because otherwise a filing system could get out of sync with the context in FileSwitch.
- A screen mode change has occurred. This means that all modules can be aware of the screen state and re-read VDU variables, for instance.

By monitoring these service calls, a module can be aware of many things that are occurring outside its control in the system.

Technical Details

Module initialisation

When RISC OS is started it automatically initialises all modules in the computer. In RISC OS 2 it does so in the order it finds modules, omitting any that are unplugged.

The way in which the kernel initialises modules has been changed in later versions of RISC OS. If there is more than one version of the same module present in the main ROM, expansion cards or extension ROMs then only the newest version of the module is initialised, where newest means the version with the highest version number.

If there are two copies of the same version, then directly executable versions (ie in main ROM or in a 32-bit wide extension ROM) are considered newer. If they are equal in this respect, then the later one in scanning order is considered to be newer.

1 The kernel first scans all modules in ROM (whether they be in the system ROM, expansion cards or extension modules), building a list of modules and their version numbers. It uses this list to determine which is the newest version of a particular module.

2 The kernel then scans down the list of modules in the system ROM. For each module in this list, the kernel initialises the newest version of that module.

Hence if an expansion card or extension ROM contains a newer version of a module in the main ROM, the kernel initialises that newer version at the point where the main ROM version would have been initialised. This allows main ROM modules to be replaced without any problems associated with initialisation order.

3 The kernel next scans down the list of modules in expansion cards. For each module in this list, the kernel initialises the newest version of that module, but with the hardware address (in R11) corresponding to that of the expansion card.

If a module is present both in the main ROM and in an expansion card, the kernel therefore initialises the newest version of that module when scanning the main ROM (as above), and then reinitialises the same module when scanning the expansion cards.

4 The kernel finally scans down the list of modules in extension ROMs. For each module in this list, the kernel checks that it is the newest version of that module, and that it has not already been initialised in lieu of a module in the main ROM or on an expansion card. If a module meets both these criteria the kernel initialises it.

Using modules

OS_Module (page 1-228) is the main application interface to modules. In its description you will find a complete list of its calls, and details of each of them.

A number of * Commands exist, most of which use OS_Module directly. Below is a table summarising OS_Module entries and the *Command equivalent.

Entry	Meaning	*Command equivalent
0	Run	*RMRun
1	Load	*RMLoad
2	Enter	module-dependent – usually provided by the module, eg *BASIC
3	ReInit	*RMReInit
4	Delete	*RMKill
5	Describe RMA	
6	Claim RMA space	
7	Free RMA space	
8	Tidy modules	*RMTidy
9	Clear	*RMClear
10	Insert module from memory	
11	As above, and move to RMA	*RMFaster (if in ROM)
12	Extract module information	*Modules & *ROMModules
13	Extend block in RMA	
14	Create new instantiation	
15	Rename instantiation	
16	Make preferred instantiation	
17	Add expansion card module	
18	Look-up module name	
19	Enumerate ROM modules	
20	Enumerate ROM modules with version	

Tidying – as mentioned above – refers to finalising all the modules, moving them together so that free RMA space is in a single block, and then re-initialising them. This solves problems with memory fragmentation.

*RMEnsure is a command that will check that a given module and version number is loaded into memory, and will execute a specified command if it is not – such as loading a module from disc (which replaces the currently loaded version, if any), or generating an error.

*Unplug will disable the ROM version of a given module. This is useful, for example, to save the RAM workspace claimed by a module you do not need to use.

*RMInsert will reverse the action of *Unplug, without initialising any modules.

Workspace

The operating system allocates one word of private workspace to each module instantiation. Normally, the module will require more and it is expected that it will use this private word as a pointer to the workspace which it claims from the RMA using OS_Module 6. Whenever the system calls a module through one of its header fields, it sets R12 to point at this private word. Hence, if this word is a pointer to workspace, the module can obtain a pointer to its true workspace by performing the instruction:

```
LDR R12,[R12]
```

The system works on the assumption that the private word is a pointer to workspace claimed in the RMA. It therefore provides suitable default actions on that basis. For example, when a module is killed the system will attempt to free any workspace claimed using this pointer, after it has called the finalisation code.

Also, the system relocates the value held in a module's workspace pointer when the RMA is 'shuffled' as a result of an RMTidy call.

Note that workspace allocated through XOS_Module will always lie on an address &xxxxxxx4. This enables code written for time-critical software (eg sound voice generators and FIQ handlers) to be aligned within the module body.

Errors in module code

Any module code which provides system extensions (SWIs and * Commands) must behave in a manner which is compatible with the operating system if an error occurs. This means that only X SWIs are called, and if anything goes wrong, the module must:

- set up R0 to point to the error block
- preserve all appropriate registers
- return with V set.

If no error has been encountered, V must be clear, and appropriate registers preserved on exit.

The above does not apply to application code within the module; this can follow any convention it wishes.

Module header format

The module indicates to the system if and where it wishes to be called by a module header. This contains offsets from the start of the module to code and information within the body of the module.

Offset	Type	Contains
&00	offset to code	start code
&04	offset to code	initialisation code
&08	offset to code	finalisation code
&0C	offset to code	service call handler
&10	offset to string	title string
&14	offset to string	help string
&18	offset to table	help and command keyword table
&1C	number	SWI chunk base number (optional)
&20	offset to code	SWI handler code (optional)
&24	offset to table	SWI decoding table (optional)
&28	offset to code	SWI decoding code (optional)

All modules must have fields up to &18. However, any of these offsets can be zero, (which means don't use this entry since the module does not contain the relevant data/code), apart from the title string. This is the offset to the zero-terminated name and if it is zero, the module cannot be referenced.

All code entries must be word aligned and inside the module code area, otherwise the checking performed by RISC OS will consider it invalid. All tables and strings must similarly be within the module or else it will be rejected.

The SWI handler fields are optional and are only used if they contain valid values.

The module header entries are described in detail in the following section of this chapter.

Service calls

Service calls are made from RISC OS to a module to indicate an occurrence of some kind. Some are claimable, and some are intended as broadcasts of the occurrence only. See the description in OS_ServiceCall (page 1-254) for a complete list of all service calls. It is followed by details of each call. Some of these service calls will also be relevant to other parts of this manual that describe modules. For example, there are service calls that are provided explicitly to serve the International module.

OS_Byte 143 is an obsolete way of calling OS_ServiceCall. It is documented, but must not be used, as it is here only for compatibility with earlier Acorn operating systems.

Module entry points

Start code

Start executing at the start point of code in a module

Offset in header

&00

On entry

R0 = pointer to command string, including module name
R12 = pointer to private word for currently preferred instantiation of the module

On exit

Doesn’t return unless error occurs.

Interrupts

Interrupts are enabled on entry
Fast interrupts are enabled

Processor Mode

Processor is in USR mode

Re-entrancy

Entry point is not re-entrant

Use

This is the offset to the code to call if the module is to be entered as the current application. An offset of zero implies that the module cannot be started up as an application, ie it is purely a service module and contains only a filing system or * Commands, etc.

This field need not actually be an offset. If it cannot be interpreted as such, ie it is not a multiple of four, or any bits are set in the top byte, then calling this field will actually execute what is assumed to be an instruction at word 0 in the module. This allows applications to have a branch at this position and hence be run directly, eg for testing. Once entered, a module may get the command line using OS_GetEnv.

Whenever the module is entered via this field, it becomes the preferred instantiation. Therefore R11 does not refer to the instantiation number.

You must exit using OS_Exit, or by starting another application without setting up an exit handler.

Start code is used by OS_Module with Run or Enter reason codes.

Initialisation Code

Set up the module, so that all other entry points are operating

Offset in header

&04

On entry

R10 = pointer to environment string (ie initialisation parameters supplied by caller of OS_Module)
R11 = I/O base or instantiation number (see below)
R12 = pointer to private word for this instantiation of the module.
If the private word ≠ 0, this implies reinitialisation after an OS_Module 8.
R13 = supervisor stack pointer

On exit

Must preserve processor mode and interrupt state
Must preserve R7 - R11 and R13
R0 - R6, R12, R14 and the flags (except V of course) can be corrupted

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

This code is called when the module is loaded and also after the RMA has been tidied (OS_Module with Tidy reason code). The module will not be called via any other entry point until this entry point has been called. Thus the initialisation code is expected to set up enough information to make all other entry points safe.

An offset of zero means that the module does not need any initialisation. The system does not provide any default actions.

The Initialisation code is used by OS_Module with Run, Load, ReInit and Tidy reason codes.

If the module is being re-entered after a OS_Module ‘tidy’, the private word may contain a non-zero value. This is the contents of the private word after the finalisation, relocated (if necessary) by the system.

Typical actions are claiming workspace (via OS_Module) and storing the workspace pointer in the private word. Other actions may include linking onto vectors, declaring the module as a filing system, etc. During initialisation, your module is not on the active module list, and so you cannot call SWIs in your own SWI chunk. Instead you must directly enter your own code.

You must not generate errors in your initialisation code. In particular, this means that you must call the error-returning (or ‘X’) form of SWIs, and must not call OS_GenerateError. For more details, see the chapter entitled *Generating and handling errors* on page 1-41.

If your module is unable to function – perhaps because of an error returned from a SWI it called – it can refuse to be initialised by returning an error in the usual way (ie by setting the V flag, and returning with R0 as an error pointer). The system removes the module and any workspace pointed to by its private word from the RMA. Note that in this case it does not call your module’s finalisation code.

The module is also passed an ‘environment string’ pointer in R10 on initialisation. This points at any string passed after the module name given to the SWI.

R11 indicates where the module has come from: if R11 = 0, then the module was loaded from the filing system or ROM or is already in memory; if R11 is > &03000000, then the module was loaded from an expansion card and R11 points at the synchronous base of the expansion card. Other values of R11 mean that the module is being reincarnated and there are <R11> other instantiations of the module.

On exit (whether or not you are returning an error), use the link register passed in R14 to return:

```
MOV PC,R14
```

Finalisation Code

Called before killing the module

Offset in header

&08

On entry

R10 = fatality indication: 0 is non-fatal, 1 is fatal
R11 = instantiation number
R12 = pointer to private word for this instantiation of the module.
R13 = supervisor stack

On exit

Must preserve processor mode and interrupt state
Must preserve R7 - R11 and R13
R0 - R6, R12, R14 and the flags can be corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

This is the reverse of initialisation. This code is called when the system is about to kill an instantiation of the module – either completely, or temporarily whilst it tidies the RMA.

If the call is fatal, the module's workspace is freed, and the workspace pointer is set to zero. If the call is non-fatal (eg the call is due to a tidy operation), the workspace (and the pointer) will be relocated by the module handler, assuming they were allocated using OS_Module's 'claim' entry.

The module is told whether the call is fatal or not by the contents of R10 as follows:

R10 = 0 means a non-fatal finalisation
R10 = 1 means a fatal finalisation

R11 contains the dynamic instantiation number: ie the position of the instantiation in the instantiation list. This will not be the same as the R11 given to initialisation. Position in the chain can vary and the length of the instantiation list can also change.

If the module generates an error on finalisation, then it remains in the RMA, and is assumed to still be initialised. The only way to remove the module from RMA in this state is by a hard reset.

If the module has no finalisation entry, its workspace is freed automatically, if the pointer contains a non-zero value.

Use link register given for normal exit. Set R0 and return with V set if refusing to die.

The module is (possibly temporarily) ‘de-linked’ when called, so you can’t, for example, execute SWIs that you recognise yourself.

Used on OS_Module with ReInit, Delete, Tidy and Clear reason codes. Also when a module of the same name is loaded the old one is killed.

Service call handler

Called when a service call is issued

Offset in header

&0C

On entry

R1 = service number
R12 = pointer to private word for this instantiation of the module
R13 = a full, descending stack

On exit

R1 can be set to zero if the service is being claimed
R0, R2 - R8 can be altered to pass back a result, depending on the service call
Registers must not be corrupted unless they are returning values.
R12 may be corrupted

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC or IRQ mode

Re-entrancy

Entry point may be re-entered by RISC OS, and so must be able to handle this

Use

This allows service calls to be recognised and acted upon. If the module does not wish to provide the service it should exit with R1 preserved. If it wishes to perform the service and to prevent other modules also performing it, it should set R1 to zero before returning, otherwise it should preserve the registers in order that other modules may have a chance to deal with the call. An offset of zero means that the module is not interested in any service calls.

It is important that you reject unrecognised service calls as quickly as possible. This example shows the recommended code to do so. It assumes the module recognises three service calls, but you can easily adapt it for other cases:

```
        TEQ R1,#Service_<1>
        TEQNE R1,#Service_<2>
        TEQNE R1,#Service_<3>
        MOVNES PC,LR                    ; reject unrecognised calls asap

        STMFD R13!,{ registers, LR }
        LDR R12,[R12]                   ; if workspace pointer required

        TEQ R1,#Service_<3>             ; now find which call we've got
        BEQ svc_3
        TEQ R1,#Service_<2>
        BEQ svc_2

svc_1   code to handle service call 1   ; if not 3 or 2, then must be 1
        LDMFD R13!,{ registers, PC }^   ; and return

svc_2   code to handle service call 2
        LDMFD R13!,{ registers, PC }^   ; and return

svc_3   code to handle service call 3
        LDMFD R13!,{ registers, PC }^   ; and return
```

Some service calls can indicate an error condition by the contents of registers on exit (the V set convention cannot be used). Others, like unknown OS_Byte, can either claim the service, in which case there is no way of indicating an error, or ignore it, in which case an error will be given (if all modules ignore it). If you want to provide things like unknown OS_Bytes, and be able to generate an error for, say, invalid parameters, you should use the OS_Byte vector instead.

Note that only R0 - R8 can be passed into a service call.

The service call handler is used when a service call is issued or via an OS_Byte 143 (page 1-226) or OS_ServiceCall (page 1-254). The service calls are described in the section on OS_ServiceCall.

Title string

Offset of a null-terminated module name

Offset in header

&10

Use

This is the offset of a null-terminated string which is used to refer to the module when OS_Module is called. The module name should be made up of alphanumeric characters and should not contain any spaces or control characters. This must be present for the module to be recognised.

Module names which contain more than one word should follow the convention of the system modules, eg ‘FileSwitch’, ‘SpriteUtils’. The case of the letters in a module name isn’t significant for the purposes of matching.

The string should be fairly short and descriptive, eg WindowManager or DiscToolkit.

The string is used by OS_Module with reason codes Delete, Enter and ReInit, and also by the *Modules, *RMEnsure and *ROMModules commands.

Help string

Used when *Help prints information from the module

Offset in header

&14

Use

This is the offset of a null-terminated string printed out by *Help before any information from the module, eg *Help Modules, *Help Commands. It is advisable that this string is present to avoid confusion. The string must not contain any control characters (except Tab, which tabs to the next multiple of eight column, or character 31 which acts as a ‘hard’ space) but may contain spaces.

To make the output of *Help Modules look neat, you should adopt the same spacing and naming conventions as the system modules. The format is as follows:

module_name Tab[*Tab*] *v.vv* (*DD MMM YYYY*)

The module name is followed by one or two Tab characters to make it appear sixteen characters long. The version number contains three digits and a full stop, eg ‘1.00’. The creation date is of the form 06 Jun 1987.

Help and command keyword table

Get help on * Commands or enter them

Offset in header

&18

Use

This table contains a list of keywords with associated help text and, in the case of commands, an entry address to the command code. Other associated data provides information on the type of command, the limits on the number of parameters it can take, etc.

It is used when OSCLI, *Status, *Configure and *Help wish to look for user-supplied keywords.

The string to match should contain only the valid characters for its entry type. For example, commands matched by OSCLI cannot contain any characters that have a special meaning in filenames. In general it is best to stick to alphanumeric characters and the ‘_’ character. The case of the letters does not matter in command matching, but should be chosen for neat output from *Help. The standard adopted by the system modules is the form ‘Echo’, ‘SetType’ etc

The table consists of a sequence of entries, terminated by a zero byte. Each entry has the following format:

String to match, null terminated
ALIGN to word boundary
Offset of code from module start, or zero if no code
Information word
Offset of invalid syntax message from module start, or zero for default message
Offset of help text from module start, or zero for no help

Figure 14.1 Format of entries in help and command keyword table

Code offset

The code offset is used for commands. A zero entry means that the string has help text only associated with it. The code is entered with these conditions:

On entry

R0 = pointer to the command tail, which you may not overwrite
R1 = number of parameters (as counted by OSCLI, which means space(s) separate parameters except within double quotation marks)
R12 = pointer to private word for this instantiation of the module
R13 = pointer to a full descending stack
R14 = return address

On exit

R0 = error pointer if anything goes wrong
R7 - R11 must be preserved

Interrupts

Interrupts are enabled on entry
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Information word

The information word contains limits on the number of parameters accepted by the command, and also 16 flags. The format is:

Byte	Contents
0	Minimum number of parameters (0 - 255)
1	OS_GSTrans map for first 8 parameters
2	Maximum number of parameters (0 - 255)
3	Flags

The command can, therefore, accept between zero and 255 parameters. OSCLI counts parameters by starting at the start of the command tail and looking for items (quoted strings or continuous characters) separated by spaces. This is why it is advisable to use spaces as parameter separators and not commas, as in commands which are compatible with the BBC series of microcomputers.

Byte 1 works as follows. Each bit corresponds to one parameter (bit zero of the byte equals the first parameter and so on). If the bit is set, the parameter is GSTrans'd before being passed on to the module. If the bit is clear, the parameter is passed directly to the module. This is useful for filing system commands which need to do filename transformations that are normally done by FileSwitch.

The flags are as follows:

Bit 31 = 1

The match string is a filing system command and is therefore only matched after OSCLI has failed to find the command in any of the module tables as a 'normal' command. OSCLI only looks at filing system commands in the filing system currently active. Commands that need this flag set are, therefore, the filing system-specific ones such as *Bye, *Logon, etc.

Bit 30 = 1

The string is to be matched by *Status and *Configure. The code in this case should scan the command tail and return a status string or set non-volatile memory as appropriate. The code is called with R0 set as follows:

R0 = 0 *Configure has been issued with no option. The module prints a syntax string and returns.

R0 = 1 *Status *option* has been issued. The module should print the currently configured status for this configuration *option*.

If R0 is neither of the above, it means that *Configure *option* has been issued; R0 is a pointer to the command tail with leading spaces skipped. The module must decode the arguments and set the configuration accordingly. If the command tail is incorrect, the module should return with V set and R0 indicating the error as follows:

R0 = 0	Bad configure option
R0 = 1	Numeric parameter needed
R0 = 2	Configure parameter too large
R0 = 3	Too many parameters
R0 > 3	R0 is a pointer to an error block for *Configure to return

Note that this facility duplicates two of the service code entries. You should use this method in preference, as the OS performs decoding of the option keywords for you.

Bit 29 = 1

*Help offset refers to a piece of code to call for that keyword, instead of the offset of a text string. The code is called with the following entry conditions:

R0 points at a buffer
R1 is the buffer length
R1 - R6 and R12 can be corrupted

On return, if R0 is non-zero, it is assumed to point at a zero-terminated string to pretty-print (see below).

Other flags

Other flags should be zero for upwards compatibility.

Invalid syntax message

The invalid syntax message is used by OSCLI as the text of an error message. If the parameters, which are given, fall outside the range specified. If a zero offset is given, a default ‘Invalid number of parameters’ error is given instead. See also Service_SyntaxError on page 1-269.

Help text

The help text is used by *Help. If a keyword in the *Help command tail matches the match string, then the help text is pretty-printed using the RISC OS internal token dictionary. Refer to OS_PrettyPrint (page 1-536) for a full list of the token dictionary.

A zero offset means no help text is to be printed. The string may contain carriage returns to force newlines. Tab (ASCII 9) is also a special character; it forces alignment to the next multiple of eight columns. Finally, ASCII 31 is a ‘hard space’, around which words lines will not be split.

SWI chunk base number

The base of chunk numbers for the module

Offset in header

&1C

Use

This offset contains the base of chunk numbers for the module. Note that it is the only offset that does not contain a pointer. RISC OS reads this offset to enable it to call the module when a SWI using its chunk range is issued.

SWI handler code

Called to handle SWIs belonging to the module

Offset in header

&20

On entry

R0 - R9 are passed from the SWI caller by RISC OS
R11 = SWI number modulo Chunk Size (ie 0 - 63)
R12 = private word pointer
R13 = supervisor stack
R14 contains the flags of the SWI caller

On exit

R0 - R9 are returned to SWI caller by RISC OS
R10 - R12 may be corrupted

Interrupts

Interrupts on entry are in the same state as when the SWI instruction was issued
Fast interrupts are enabled

Your code should not explicitly enable interrupts, as if they are disabled on entry, there is likely to be a good reason for this – such as your SWI being called from an interrupt handling routine. If you do need to disable IRQs during an atomic operation, you should only do so for the minimum time possible, and should afterwards restore them to their previous state. Recommended code to do so is:

```
MOV     Rn, PC
ORR     Rm, Rn, #I_bit          ; #I_bit is 1<<27
TEQP    Rm, #0                  ; disable interrupts
…
Your code for short atomic operation        ; must preserve Rn
…
TEQP    Rn, #0                  ; restore interrupts
```

The TEQP instructions are not changing processor mode, but merely disabling IRQs in the current mode, so need not be followed by a no-op.

Processor Mode

Processor is in SVC mode

Re-entrancy

Your module may issue SWIs to itself; if it does, it must handle them

Use

These entries allow a module to ask to be given a range of otherwise unrecognized SWIs. The SWI chunk number is the base of the range to be intercepted. SWIs in the range:

base to (base + SWI chunk size – 1)

are passed to the handler code. The module SWI chunk size is defined by the operating system to be &40 (64). For example, this entry in the Wimp module is &400C0, implying that it can accept SWIs in the range &400C0 - &400FF.

These fields are optional; if they contain implausible values, the system will ignore them. The checks made are:

- base is a multiple of the chunk size and has a 0 top byte
- code offset is a multiple of four with the top six bits zero.

See the section entitled *SWI numbers in detail* on page 1-26 for more details on SWI and chunk numbers.

When the SWI handler code is called, the SWI number reduced to the range 0 to (chunk size – 1) is passed in R11. The module then checks whether it is one which it recognises and if so, deals with it appropriately. The suggested code for doing this is:

```
.SWIentry
        LDR     R12, [R12]                      ; get workspace pointer
        CMP     R11, #(EndOfJumpTable – JumpTable)/4
        ADDLO   PC, PC, R11, LSL #2             ; dispatch if in range
        B       UnknownSWIerror                 ; unknown SWI
```

```
.JumpTable
        B       MySWI_0
        B       MySWI_1
        ...............
        B       MySWI_n
.EndOfJumpTable
.UnknownSWIError
        ADR     R0, ErrToken
        MOV     R1, #0
        MOV     R2, #0
        ADR     R4, ModuleTitle                 ; From module header
        SWI     "XMessageTrans_ErrorLookup"
        ORRS    PC, R14, #Overflow_Flag
.ErrToken
        EQUD    &1E6                            ; Same as system message
        EQUS    "BadSWI"                        ; Token to look up
        EQUB    0
        ALIGN
```

Note that the address calculation on the PC to jump to the appropriate branch instruction relies on there being exactly one instruction between the ADDLO and the B MySWI_0 instruction.

The R14 given to the SWI code contains the flags of the SWI caller, except that V has been cleared. So, to return without updating the flags, use

```
MOVS PC, R14
```

Otherwise alter the link register, for example by executing

```
ORRS PC, R14, #Overflow_Flag
```

Note that all the flags returned to the system are returned to the caller, so user's conditional code must be written with this in mind.

Bit 17 in the given SWI number is not significant. The code is called on the assumption that it is the 'bit 17 set' version of the SWI. This means that the code must set R0 and return V set on encountering an error. Any error is then automatically dealt with by the system if the user actually asked for the 'bit 17 clear' version.

SWI decoding table

Pointer to table of SWI names

Offset in header

&24

Use

When the SWIs OS_SWINumberFromString and OS_SWINumberToString are called, there are two ways that the conversion can occur. If the table pointed to by this offset contains the string for the required entry, then that is used. If it isn't there and the table pointer is 0, then the following offset is called, to allow the module code to perform the conversion.

The table format is:

SWI group prefix
Name of 0th SWI
Name of 1st SWI
…
…
Name of *n*th SWI
0 byte to terminate

All names are null terminated. The group prefix is the first part of the full SWI name: ie the first SWI's full name is *GroupPrefix_NameOf1st*. For example, ShellCLI's table is:

```
EQUS    "Shell"
EQUB 0
EQUS    "Create"
EQUB 0
EQUS    "Destroy"
EQUB 0
EQUB 0
```

In this example, the chunk base number is &405C0. The SWI &405C1 would therefore be converted into 'Shell_Destroy' if passed to OS_SWINumberToString.

The OS adds an 'X' if the SWI has bit 17 set, followed by the group prefix, followed by '_', then the individual SWI name. If the table does not contain enough entries, then the SWI name field is filled in by the offset from the chunk base (in decimal).

If the table field is zero, then the code field is used (see below). This field is also used when converting from strings to numbers.

SWI decoding code

Entry for code to convert to and from SWI number and string

Offset in header

&28

On entry

R12 = private word pointer
R13 = supervisor stack
R14 = return address

Text to number

R0 = any number less than zero
R1 = pointer to the string to convert (terminated by a control character)

Number to text

R0 = SWI number ANDed with 63: ie offset within module's chunk
R1 = pointer to output buffer
R2 = offset within output buffer at which to place the text
R3 = size of buffer

On exit

R12 preserved

Text to number

R0 = offset into chunk (0 - 63) if SWI recognised, <0 otherwise
R1 - R6 preserved

Number to text

R0 preserved
R1 preserved
R2 = updated by length of text
R3 - R6 preserved

Interrupts

Interrupts are enabled on entry
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

This entry is used where a SWI name is not defined in the SWI decode table. If it cannot be decoded, and the table pointer is 0, then return with the registers unchanged and RISC OS will provide a suitable default.

When converting from number to text, RISC OS will append a null at the position after the length you returned.

SWI Calls

OS_Byte 143 (SWI &06)

Issue module service call

On entry

R0 = 143
R1 = service type
R2 = argument for service

On exit

R0, R1 preserved
R2 = may contain a return argument

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 with the BBC series of microcomputers, and is used for calling the modules' service entries. Only OS_ServiceCall should be used in new code.

Related SWIs

OS_ServiceCall (page 1-254)

Related vectors

ByteV

OS_Module (SWI &1E)

Perform a module operation

On entry

R0 = reason code
other registers are parameters and depend upon the reason code

On exit

R0 preserved
other register states depends on the reason code

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI provides a number of calls to manipulate modules. The value in R0 describes the operation to perform as below:

R0	Meaning	Page
0	Run	1-231
1	Load	1-232
2	Enter	1-233
3	ReInit	1-234
4	Delete	1-235
5	Describe RMA	1-236
6	Claim	1-237
7	Free	1-238

R0	Meaning	Page
8	Tidy	1-239
9	Clear	1-240
10	Insert module from memory	1-241
11	Insert module from memory and move into RMA	1-242
12	Extract module information	1-243
13	Extend block	1-244
14	Create new instantiation	1-245
15	Rename instantiation	1-246
16	Make preferred instantiation	1-247
17	Add expansion card module	1-248
18	Lookup module name	1-249
19	Enumerate ROM modules	1-250
20	Enumerate ROM modules with version	1-252

This call performs simple checks when deleting and moving modules. These actions give an error if the system ‘thinks’ you are applying them to a module currently active, for example, if you try to *RMKill BASIC from within BASIC.

This check is applied whenever the system is about to call a module’s finalise entry. Hence simple applications need not keep checks on this explicitly. More complex modules which, for example, run subtasks, need to keep their own state checks in order to avoid being removed when they are due to be returned to at some point.

Many of the OS_Module calls refer to a module title. This has some general restrictions. The name passed is terminated by any control character or space and can be abbreviated with a full stop. For example, ‘Eco.’ is an abbreviation for ‘Econet’. The title field in the module is similarly terminated by control characters and spaces. The pattern matching ignores the case of both strings, and allows any characters other than space or full stop. You should restrict your titles, however, to alphanumerics and ‘_’ for future compatibility.

As usual, errors are indicated by V being set and an error pointer in R0. These errors may be generated by one of the modules, and the error block addressed by R0 might reside in a module’s code. You should therefore not rely on the error block remaining in the same place across calls to OS_Module.

As the checks within this call cannot tell which instantiation of a module is active, no instantiation may die when one of them is the current application. The module name can also have an instantiation postfix. This consists of ‘%’ followed by the instantiation name. This name field can be abbreviated in the same way as the module name. If no instantiation is given, the currently preferred instantiation is referenced.

In the following pages, the reason codes for this command are fully explained. The details of general SWI operation are as per this description.

Related SWIs

None

Related vectors

None

OS_Module 0 (SWI &1E)

Run

On entry

R0 = 0 (reason code)
R1 = pointer to pathname plus optional parameters

On exit

Does not return unless error occurs

Use

This call is equivalent to loading, initialising, and then entering the module. If the module can be started as an application, it will be, and so the call will not return.

Possible errors from RISC OS are ‘File '*filename*' not found’, ‘Not enough memory in module area’ (or, under RISC OS 2, ‘No room in RMA’), ‘This is not a relocatable module’. The module itself may return errors such as ‘Duplicate module refused to die’, and ‘Module refuses to initialise’.

Related reason codes

1, 2

OS_Module 1 (SWI &1E)

Load

On entry

R0 = 1 (reason code)
R1 = pointer to pathname and optional parameters

On exit

R0, R1 preserved

Use

This reason code attempts to claim a block of the RMA and *Loads the file if it has the correct file type of &FFA. The header fields of the module are then checked for validity.

If another module has the same name, it attempts to kill the duplicate module. This will give an error if the module refuses to die. Note that this allows system modules to be upgraded with new versions simply by loading the new version. All instantiations of the duplicate are killed.

It sets the private workspace word to 0, calls the module through its initialise address and links it to the end of the module list, or replaces the old module of the same name. The module is initialised as instantiation ‘Base’.

The filename should be terminated suitably for OS_File. The terminator can be space, in which case there can be a parameter string after the filename to pass to the module initialisation.

Possible errors are ‘File '*filename*' not found’, ‘Not enough memory in module area’ (or, under RISC OS 2, ‘No room in RMA’), ‘This is not a relocatable module’, and other errors dependent on the module, such as ‘Duplicate module refused to die’, and ‘Module refuses to initialise’.

Related reason codes

0, 2

OS_Module 2 (SWI &1E)

Enter

On entry

R0 = 2 (reason code)
R1 = pointer to module name
R2 = pointer to parameters

On exit

Does not return unless error occurs

Use

If the module doesn’t have a start address, then this call simply returns. If it does, this call resets the supervisor stack pointer to the top of the stack, sets user mode and enters the module, hence making it the current application. Any specified instantiation will become the preferred instantiation. The possible error is ‘Module '*modulename*' not found’.

For a description of how a module is started up as an application, refer to OS_FSControl 2 (page 2-85).

Related reason codes

0

OS_Module 3 (SWI &1E)

Reinitialise

On entry

R0 = 3 (reason code)
R1 = pointer to module name plus any parameters for initialisation

On exit

R0, R1 preserved

Use

This is equivalent to reloading the module. It is intended for use in forcing modules that have become confused into a sensible state, without having to reload them explicitly from the filing system. The instruction calls the module through its finalise address and deletes any workspace. It then calls it through its initialisation address to reinitialise it. If the module fails to initialise it is removed from the RMA. Possible errors are ‘Module '*modulename*' not found’, and others dependent on the module.

Related reason codes

8, 9

OS_Module 4 (SWI &1E)

Delete

On entry

R0 = 4 (reason code)
R1 = pointer to module name

On exit

R0, R1 preserved

Use

This reason code (and *RMKill) kill off the currently preferred instantiation of the module or the one specified in the name. For example:

```
*RMKill FileCore%Base
```

This calls the module through its finalise address, frees any workspace pointed at by the private word, delinks the module from the module list and frees the space it was occupying. Possible errors are ‘Module not found’ and others dependent on the module.

Related reason codes

None

OS_Module 5 (SWI &1E)

Describe RMA

On entry

R0 = 5 (reason code)

On exit

R0 preserved
R2 = size of largest block available in bytes
R3 = total amount free in RMA in bytes

Use

This call returns information on the state of the RMA. It does this by calling OS_Heap with the appropriate descriptor.

Related reason codes

6

OS_Module 6 (SWI &1E)

Claim

On entry

R0 = 6 (reason code)
R3 = required size

On exit

R0 preserved
R2 = pointer to claimed block
R3 preserved

Use

This calls the heap manager to claim workspace in the RMA. If it fails and application workspace is not currently being used then it will attempt to reallocate this memory and retry. It returns with V set if it is still unsuccessful. This call is useful for claiming workspace during the module's initialisation, but may also be used from other module entries.

The possible error is 'Not enough memory in module area' (or, under RISC OS 2, 'No room in RMA').

Related reason codes

5, 7

OS_Module 7 (SWI &1E)

Free

On entry

R0 = 7 (reason code)
R2 = pointer to block

On exit

R0 preserved
R2 preserved

Use

This calls the heap manager to free a block of workspace claimed from the RMA.

The possible error is ‘Not a heap block’.

Related reason codes

6

OS_Module 8 (SWI &1E)

Tidy

On entry

R0 = 8 (reason code)

On exit

R0 preserved

Use

This gives each instantiation of all modules in turn, from the end of the module list and working backwards, a non-fatal finalisation call. Instantiations of a particular module are killed in the order they appear on the current instantiation list.

Should any instantiation of any module refuse to die, then any modules which have already been killed are re-initialised. Should any of these give an error during re-initialisation, they are then deleted from the system. The SWI then exits with the original error returned by the module that first refused to die.

If all modules die successfully, this call then collects the RMA together into one large unfragmented block, and reinitialises the modules again. Any private words containing pointers to workspace blocks in the RMA are relocated. This should enlarge application space.

Related reason codes

3, 9

OS_Module 9 (SWI &1E)

Clear

On entry

R0 = 9 (reason code)

On exit

R0 preserved

Use

This deals with each module in turn, removing it from the module list and calling it through its finalise address, if it isn't a ROM module. Errors are generated if modules fail to die.

Related reason codes

3, 8

OS_Module 10 (SWI &1E)

Insert module from memory

On entry

R0 = 10 (reason code)
R1 = pointer to start of module

On exit

R0, R1 preserved

Use

This takes a pointer to a block of memory and links it into the module chain, without moving it. Header fields are checked for validity. All duplicate modules are killed. If it is successful, then the module is called at its initialisation entry.

Possible errors are ‘Duplicate module refuses to die’ and ‘Module refuses to initialise’.

The word immediately before the module start (ie at address R1–4) must contain the length of the module in bytes.

Related reason codes

11

OS_Module 11 (SWI &1E)

Insert module from memory and move into RMA

On entry

R0 = (reason code)
R1 = pointer to start of module
R2 = length of module in bytes

On exit

R0 - R2 preserved

Use

This takes a pointer to a block of memory, and checks its header fields for validity. It then kills any duplicate module, copies the block into the RMA, initialises it and links it into the module chain.

Possible errors are ‘Duplicate module refuses to die’, ‘No room in RMA’ and ‘Module refuses to initialise’.

Related reason codes

10

OS_Module 12 (SWI &1E)

Extract module information

On entry

R0 = 12 (reason code)
R1 = module number, or 0 for first call
R2 = instantiation number, or 0 for all

On exit

R0 preserved
R1 = updated module number
R2 = updated instantiation number
R3 = module base
R4 = private word (usually workspace pointer)
R5 = pointer to instantiation postfix

Use

This returns pointers to modules and the contents of their private word. It searches the list of modules to see if the module pointer given in R1 is valid. If it is valid, the next descriptor in the module chain is referenced, otherwise the first module descriptor is referenced. Information from the referenced descriptor is then returned. The information returned is exactly that printed by the *Modules command.

Specifying the instantiation number and index in the module list allows all module instantiations to be enumerated. Enumeration can be started with 0 in R1 and R2. This call will:

- count down the module list to find the R1th entry; error if list runs out
- count down the instantiation list to R2th entry; error if list runs out
- set up return information

If the module has more instantiations, R2 += 1 else R1 += 1, R2 = 0

Possible errors are ‘No more modules’ or ‘No more incarnations of that module’.

Related reason codes

13

OS_Module 13 (SWI &1E)

Extend block

On entry

R0 = 13 (reason code)
R2 = pointer to workspace block
R3 = change in size in bytes

On exit

R0 preserved
R2 = pointer to new allocated block
R3 preserved

Use

This allows modules to extend workspace blocks claimed in the RMA. It calls OS_Heap with the appropriate descriptor and attempts to enlarge the RMA if this fails.

The possible error is ‘No room in RMA’.

Related reason codes

12

OS_Module 14 (SWI &1E)

Create new instantiation

On entry

R0 = 14 (reason code)
R1 = pointer to new instantiation name and any parameters for initialisation

On exit

R0, R1 preserved

Use

This creates new instantiations of existing modules, using the syntax:

`module_title%instantiation`

For example:

```
FileCore%RAM
```

Related reason codes

15, 16

OS_Module 15 (SWI &1E)

Rename instantiation

On entry

R0 = 15 (reason code)
R1 = pointer to current *module%instantiation* name
R2 = pointer to new instantiation name

On exit

R0 - R2 preserved

Use

This renames an existing instantiation of a module. For example:

```
FileCore%RAM
```

to

```
FileCore%ADFS
```

Related reason codes

14, 16

OS_Module 16 (SWI &1E)

Make preferred instantiation

On entry

R0 = 16 (reason code)
R1 = pointer to *module%instantiation* name

On exit

R0, R1 preserved

Use

This enables you to select the preferred instantiation of a particular module.

Related reason codes

14, 15

OS_Module 17 (SWI &1E)

Add expansion card module

On entry

R0 = 17 (reason code)
R1 = pointer to environment string
R2 = chunk number
R3 = ROM section

On exit

R0 - R3 preserved

Use

This allows expansion card and extension ROM modules to be added to the module list. Note that extension ROMs are not supported in RISC OS 2.

Valid ROM sections are:

ROM section	Meaning	
–1	System ROM	
0	Expansion card 0	
1	Expansion card 1	
2	Expansion card 2	
3	Expansion card 3	
–2	Extension ROM 1	(not in RISC OS 2)
–3	Extension ROM 2	(not in RISC OS 2)
–4	Extension ROM 3 (etc)	(not in RISC OS 2)

Related reason codes

10

OS_Module 18 (SWI &1E)

Look-up module name

On entry

R0 = 18 (reason code)
R1 = pointer to full *module_title*%*instantiation* name

On exit

R0 preserved
R1 = module number
R2 = instantiation number
R3 = pointer to module code
R4 = private word contents
R5 = pointer to postfix string

Use

This returns pointers to modules and the contents of their private word. It searches the list of modules to see if the module pointer given in R1 is valid. If it is valid, the module descriptor is referenced. Information from the referenced descriptor is then returned.

Related reason codes

12, 19, 20

OS_Module 19 (SWI &1E)

Enumerate ROM modules

On entry

R0 = 19 (reason code)
R1 = module number (0 to start full enumeration)
R2 = ROM section (–1 to start full enumeration)

On exit

R0 preserved
R1 = module number of found module + 1
R2 = ROM section of found module
R3 = pointer to module name
R4 = –1 unplugged
0 inserted but not in the module chain ie dormant
1 active
2 running
R5 = chunk number of expansion card or extension ROM module

Use

This call returns information on one module that is currently in ROM, along with its status. The module found is the given number of modules on from the start of the given ROM section. If there are insufficient modules in the ROM section then the search continues with the next section; so the fifth module in a four module section would in fact be the first module of the next section.

The ROM sections are scanned in this order:

ROM section	Meaning	
–1	System ROM	
0	Expansion card 0	
1	Expansion card 1	
2	Expansion card 2	
3	Expansion card 3	
–2	Extension ROM 1	(not in RISC OS 2)
–3	Extension ROM 2	(not in RISC OS 2)
–4	Extension ROM 3 (etc)	(not in RISC OS 2)

The values returned in R0 - R2 are the correct ones to use this call to enumerate the next module; hence repeated calls will give a full enumeration of all ROM modules.

The call returns the error ‘No more modules’ (error number &107) if there are no more modules from the point specified in the ordering.

Related reason codes

12, 18, 20

OS_Module 20 (SWI &1E)

Enumerate ROM modules with version

On entry

R0 =20 (reason code)
R1 = module number (0 to start full enumeration)
R2 = ROM section (–1 to start full enumeration)

On exit

R0 preserved
R1 = module number of found module + 1
R2 = ROM section of found module
R3 = pointer to module name
R4 = –1 unplugged
0 inserted but not in the module chain ie dormant
1 active
2 running
R5 = chunk number of expansion card or extension module
R6 = BCD version number (derived from module's help string)

Use

This call returns information on one module that is currently in ROM, along with its status. The call is identical to OS_Module 19, except that on exit R6 holds a BCD (binary coded decimal) form of the module's version number, as derived from the module's help string. The top 16 bits of this value hold the integer part of the version number, and the bottom 16 bits hold the fractional part: eg if the version number of the module is '3.14' then the value returned would be &00031400.

The module found is the given number of modules on from the start of the given ROM section. If there are insufficient modules in the ROM section then the search continues with the next section; so the fifth module in a four module section would in fact be the first module of the next section.

The ROM sections are scanned in this order:

ROM section	Meaning	
–1	System ROM	
0	Expansion card 0	
1	Expansion card 1	
2	Expansion card 2	
3	Expansion card 3	
–2	Extension ROM 1	(not in RISC OS 2)
–3	Extension ROM 2	(not in RISC OS 2)
–4	Extension ROM 3 (etc)	(not in RISC OS 2)

The values returned in R0 - R2 are the correct ones to use this call to enumerate the next module; hence repeated calls will give a full enumeration of all ROM modules.

The call returns the error ‘No more modules’ (error number &107) if there are no more modules from the point specified in the ordering.

Related reason codes

12, 18, 19

Service Calls

OS_ServiceCall (SWI &30)

Issue a service call to a module

On entry

R1 = service number
other registers are parameters and depend upon the service number

On exit

R1 = 0 if service was claimed, preserved otherwise
other registers up to R8 may be modified if the service was claimed

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ServiceCall is used to issue a service call. It can be used by any program (including a module) which wishes to pass a service around the current module list. For example, someone wishing to use FIQs might issue the claim/release service calls.

A module may claim a service call by setting R1 to 0 on exit; this prevents the service call from being passed to any further modules. There are some service calls that you must not claim, since it is essential that all modules receive them; such cases are clearly documented.

Here is a list of the service calls available to application writers, the details of which can be found on the specified pages. The names given below omit the initial 'Service_' for clarity:

No	Name	Meaning	Page
&04	UKCommand	Unknown command	1-259
&06	Error	Error has occurred	1-260
&07	UKByte	Unknown OS_Byte	1-261
&08	UKWord	Unknown OS_Word	1-262
&09	Help	*Help has been called	1-263
&0B	ReleaseFIQ	Release FIQ	1-136
&0C	ClaimFIQ	Claim FIQ	1-137
&11	Memory	Memory controller about to be remapped	3-66
&12	StartUpFS	Start up filing system	2-24
&27	Reset	Post-Reset	3-67
&28	UKConfig	Unknown *Configure	1-264
&29	UKStatus	Unknown *Status	1-265
&2A	NewApplication	Application about to start	1-266
&40	FSRedeclare	Start up filing system	2-25
&41	Print	For internal use only	3-607
&42	LookupFileType	Look up file type	1-267
&43	International	International service	3-771
&44	Keyhandler	Keyboard handler	1-879
&45	PreReset	Pre-reset	4-135
&46	ModeChange	Mode change	1-638
&47	ClaimFIQinBackground	Claim FIQ in background	1-138
&48	ReAllocatePorts	Econet restarting	2-652
&49	StartWimp	Start up any resident module tasks using Wimp_StartTask	3-68
&4A	StartedWimp	Request to task modules to set `taskHandle` variable to zero	3-70
&4B	StartFiler	Request to filing-system-specific desktop filers to start up	2-500
&4C	StartedFiler	Request to filing-system-specific desktop filers to set `taskHandle` variable to zero	2-502
&4D	PreModeChange	Mode change	1-639

No	Name	Meaning	Page
&4E	MemoryMoved	OS_ChangeDynamicArea has just finished	1-367
&4F	FilerDying	Notification that the Filer module is about to close down	2-503
&50	ModeExtension	Allow soft modes	1-641
&51	ModeTranslation	Translate modes for unknown monitor types	1-645
&52	MouseTrap	The Wimp has detected a significant mouse movement	3-72
&53	WimpCloseDown	Notification that the Window Manager is about to close down a task	3-73
&54	Sound	Parts of the Sound system are starting or dying	4-17
&55	NetFS	Either a *Logon, a *Bye or a *SDisc/*Mount has occurred	2-348
&56	EconetDying	Econet is about to leave	2-653
&57	WimpReportError	Request to suspend trapping of VDU output so an error can be displayed	3-75
&59	ResourceFSStarted	The file structure inside ResourceFS has changed	2-419
&5A	ResourceFSDying	ResourceFS is killed	2-420
&5B	CalibrationChanged	Screen calibration is changed	3-342
&5C	WimpSaveDesktop	Save some state to a desktop boot file	3-76
&5D	WimpPalette	Palette change	3-77
&5E	MessageFileClosed	Message files have been closed	3-749
&5F	NetFSDying	NetFS is dying	2-349
&60	ResourceFSStarting	ResourceFS module is reloaded or reinitialised	2-421
&64	TerritoryManagerLoaded	Tell territory modules to register themselves.	3-798
&65	PDriverStarting	PDriver sharer module started	3-608
&66	PDumperStarting	PDriverDP module starting up	3-686
&67	PDumperDying	PDriverDP module dying	3-687
&68	CloseFile	Close an object, and any children of that object	2-26

No	Name	Meaning	Page
$69	IdentifyDisc	Identify disc format	2-220
&6A	EnumerateFormats	Enumerate available disc formats	2-504
&6B	IdentifyFormat	Identify disc format name	2-281
&6C	DisplayFormatHelp	Display list of available formats	2-282
&6D	ValidateAddress	OS_ValidateAddress has been called with an unrecognised area	1-368
&6E	FontsChanged	New Font$Path detected	3-425
&6F	BufferStarting	Notifies modules that the buffer manager is starting	4-87
&70	DeviceFSStarting	DeviceFS is starting	2-431
&71	DeviceFSDying	DeviceFS is dying	2-432
&72	SwitchingOutputToSprite	Output switched to sprite, mask or screen	1-787
&73	PostInit	All modules have been initialised	1-268
&75	TerritoryStarted	New territory starting	3-799
&76	MonitorLeadTranslation	Translate monitor lead ID	1-646
&78	PDriverGetMessage	Get common messages file	3-609
&79	DeviceDead	Device has been killed by DeviceFS	2-433
&7A	ScreenBlanked	Screen blanked by screen blanker	4-110
&7B	ScreenRestored	Screen restored by screen blanker	4-111
&7C	DesktopWelcome	Desktop starting	3-78
&7D	DiscDismounted	Disc dismounted	2-506
&7E	ShutDown	Switcher shutting down	3-79
&7F	PDriverChanged	Currently selected printer driver has changed	3-610
$80	ShutdownComplete	Shutdown completed	3-80
&81	DeviceFSCloseRequest	Opening a device which already has the maximum number of streams open	2-434
&82	InvalidateCache	Broadcast whenever the cache is flushed within ColourTrans	3-343
&83	ProtocolDying	Part of the AUN Driver Control Interface	2-654
&84	FindNetworkDriver	Part of the AUN Driver Control Interface	2-655
&85	WimpSpritesMoved	Wimp sprite pools have moved	3-75

No	Name	Meaning	Page
&86	WimpRegisterFilters	Allows the Filter Manager to install filters with the Window Manager	3-75
&87	FilterManagerInstalled	Filter Manager starting up	3-302
&88	FilterManagerDying	Filter Manager dying	3-303
&89	ModeChanging	Mode change	1-648
&8A	Portable	Power down or up	4-207
&8B	NetworkDriverStatus	Part of the AUN Driver Control Interface	2-656
&8C	SyntaxError	Syntax error translation	1-269
&10800	ADFSPodule	Issued by ADFS to locate an ST506 expansion card	4-136
&10801	ADFSPoduleIDE	Issued by ADFS to locate an IDE expansion card	4-137
&10802	ADFSPoduleIDEDying	IDE expansion card dying	4-138

Related SWIs

OS_Byte 143 (page 1-226)

Related vectors

None

Service_UKCommand (Service Call &04)

Unknown command

On entry

R0 = pointer to command
R1 = &04 (reason code)

On exit

R0 = 0 if command performed with no error, or error pointer if command performed with error, or preserved to pass on
R1 = 0 to claim the command, or preserved to pass on

Use

This service call is issued when a command is unknown. It is issued after OSCLI has searched modules, but before the filing system is called to try to *Run the command. It is also used to implement NetFS file server commands.

If your module recognises the command, you should try to execute it, claiming the service call by setting R1 to 0 as usual. If the command was successful you should set R0 to 0 when claiming the call; if an error occurred you should instead set R0 to point to the error buffer.

Note that this is the ‘historical’ way of dealing with unknown commands. You should, in preference, use the command string entry point.

Service_Error (Service Call &06)

Error has occurred

On entry

R0 = pointer to error block
R1 = &06 (reason code)

On exit

R0 preserved
R1 preserved to pass on (must never be claimed)

Use

This call is issued after an error has occurred but before the error handler is called. It is included ‘for your information’, and must not be claimed.

Service_UKByte (Service Call &07)

Unknown OS_Byte

On entry

R1 = &07 (reason code)
R2 = OS_Byte number (ie value in R0 when OS_Byte was called)
R3 = first parameter (ie value in R1 when OS_Byte was called)
R4 = second parameter (ie value in R2 when OS_Byte was called)

On exit

R1 = 0 to claim, else preserved to pass on
R3 = value to return in R1 to caller
R4 = value to return in R2 to caller
Errors cannot be returned

Use

If the OS_Byte number is one used by your module, you should execute it and claim the call by setting R1 to zero.

If you don't recognise the OS_Byte number, pass the call on by returning with the registers preserved.

This method of adding OS_Byte calls is deprecated, and you should instead claim the ByteV software vector. See the chapter entitled *Software vectors* on page 1-63.

Service_UKWord (Service Call &08)

Unknown OS_Word

On entry

R1 = &08 (reason code)
R2 = OS_Word number (ie value in R0 when OS_Word was called)
R3 = pointer to OS_Word parameter block (ie value in R1 when OS_Word was called)

On exit

R1 = 0 to claim, else preserved to pass on
Errors cannot be returned

Use

If the OS_Word number is one used by the module it is passing through, you should execute it and claim the call by setting R1 to zero.

If you don't recognise the OS_Word number, pass the call on by returning with the registers preserved.

This method of adding OS_Word calls is deprecated, and you should instead claim the WordV software vector. See the chapter entitled *Software vectors* on page 1-63.

Service_Help (Service Call &09)

*Help has been called

On entry

R0 = pointer to command
R1 = &09 (reason code)

On exit

R0 preserved
R1 = 0 to claim, else preserved to pass on

Use

This is issued at the start of *Help. You should claim this call only if you wish to replace *Help completely. The usual way for a module to provide help is through its help text table.

Service_UKConfig (Service Call &28)

Unknown *Configure

On entry

R0 = pointer to command tail, or 0 if none given
R1 = &28 (reason code)

On exit

R0 < 0 for no error, or a small integer for errors described below,
or error pointer for other errors
R1 = 0 if configure option recognised and no error, else preserved to pass on

Use

If R0 = 0 on entry, you should print your *Configure syntax line(s), if any, and exit with registers preserved.

If R0 ≠ 0, then R0 is a pointer to the command tail. If you decode the command tail, and recognise it, you should claim the call by setting R1 to 0. If an error is detected, should also return with V set and return the error in R0 as follows:

Value	Meaning
0	Bad *Configure option
1	Numeric parameter needed
2	Parameter too large
3	Too many parameters
>3	R0 is an error pointer returned by *Configure

If you don't recognise the command tail, you should exit with registers preserved.

Note that it is also possible to trap unknown *Configure commands through the module's command table (see the section entitled *Help and command keyword table* on page 1-216) – which is the preferred method. Only one of these mechanisms should be used.

Service_UKStatus (Service Call &29)

Unknown *Status

On entry

R0 = pointer to command tail, or 0 if none given
R1 = &29 (reason code)

On exit

R0 preserved
R1 = 0 is status option recognised and no error, else preserved to pass on

Use

If R0 = 0, you should list your status(es) and pass on the service call.

If R0 ≠ 0, then R0 is a pointer to the command tail. If you decode the command tail, and recognise it, you should print the associated information and claim the call. Otherwise you should not claim the call.

Note that it is also possible to trap unknown *Status commands through the module's command table (see the section entitled *Help and command keyword table* on page 1-216) – which is the preferred method. Only one of these mechanisms should be used.

Service_NewApplication (Service Call &2A)

Application about to start

On entry

R1 = &2A (reason code)

On exit

R1 = 0 to prevent application from starting, else preserved to pass on

Use

This service is called when an application is about to start due to a *Go, *RMRun or *Run-type operation. If you don’t want the application to start, you should claim the call, otherwise pass it on.

Service_LookupFileType (Service Call &42)

Look up file type

On entry

R1 = &42 (reason code)
R2 = file type (in lower three nibbles)

On exit

R1 = 0 if the module knows the file type, else preserved to pass on
R2 = first four characters, if known, else preserved
R3 = last four characters, if known, else preserved

Use

This call is passed round when FileSwitch is unable to convert a hexadecimal 3-digit file type *xxx* into a textual name, because it is unable to find the system variable File$Type_*xxx*. If the file type passed in R2 is known to you, you should return with R1=0, and R2, R3 containing the eight characters in the name. If no-one claims the call, FileSwitch will convert the number into a three-digit hex value padded with spaces. This might be loaded as follows:

```
        ADR     R1, nameString
        LDMIA   R1, {R2,R3}
        MOV     R1, #0
        MOV     PC, R14
.nameString
        EQUS    "My Type "              ; String must be eight bytes long
```

Service_PostInit (Service Call &73)

All modules have been initialised

On entry

—

On exit

This service call should not be claimed.

Use

This is issued on a reset, after all the ROM resident modules (including those on extension ROMs and expansion cards) have been initialised.

Service_SyntaxError (Service Call &8C)

Syntax error translation

On entry

R1 = &8C (reason code)
R2 = pointer to the issued command's code offset in the module's help and command keyword table (ie R2+4 is the command's information word; see page 1-216)
R3 = base address of module providing command
R4 = pointer to command string in module

On exit

R0 = pointer to error block giving new syntax message, else preserved to pass on
R1 = 0 to claim service call, else preserved to pass on
R2 - R4 preserved

Use

This service call is issued just before a syntax error is returned from a module * Command. It is provided so that modules can localise their error messages for a particular territory.

On entry, the registers hold sufficient information to identify the particular command being issued, and in which module it resides.

If the service call is claimed, RISC OS outputs the returned error string in the block pointed to by R0; otherwise it uses the syntax error message in the module's help and command keyword table (see page 1-216).

This service call is only issued by RISC OS 3 (version 3.10) or later.

*Commands

*Modules

Displays information about all installed relocatable modules

Syntax

```
*Modules
```

Parameters

None

Use

*Modules displays information about all relocatable modules which are currently installed in the machine.

The command displays the number allocated to each module, its position in memory, the address of its workspace, and its name.

- The number may change as other modules are installed and removed.
- The names listed by this command are the module titles, which are used as parameters for other commands such as *RMKill.

Example

```
*Modules
No. Position Workspace Name
  1 0380BED8 00000000  UtilityModule
  2 038251A8 01800014  Podule
    …
    …
 81 039EAF10 00000000  !Edit
 82 039F17E4 0181E984  DOSFS
```

Related commands

*ROMModules

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*RMClear

Deletes all relocatable modules from the module area

Syntax

```
*RMClear
```

Parameters

None

Use

*RMClear deactivates all relocatable modules in the module area, deletes them, and frees their workspace. Use this command only with extreme caution, as it is so drastic in its effects.

ROM resident modules are not affected by *RMClear; if you wish to disable such a module, you should use *RMKill or *Unplug.

Related commands

*RMInsert, *RMKill, *RMReInit, *RMTidy, *Unplug

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*RMEnsure

Checks the presence and version of a module

Syntax

```
*RMEnsure module_title version_number [command]
```

Parameters

module_title	the title of any currently installed module
version_number	a number against which the version number will be checked
command	a Command Line command

Use

*RMEnsure checks that a module is present and is the given version (or a more recent one). A command, optionally given as a third parameter, is executed if this is not the case, or – if none is specified – an error is generated. *RMEnsure is usually used in command scripts or programs to ensure that modules they need are loaded and of a recent enough version.

Example

```
*RMEnsure WindowManager 2.01 *RMLoad System:Wimp
```

Related commands

None

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*RMFaster

Makes a module faster by copying it from ROM to RAM

Syntax

`*RMFaster` *`module_title`*

Parameters

`module_title` the title of any ROM resident module

Use

*RMFaster makes a copy of a ROM resident relocatable module and places it in RAM. The module will run faster because RAM can be accessed faster than ROM.

In doing so, the module is moved to the end of the module list. You should be aware that this can cause problems later; for example, the relative ordering of some modules is important to the *RMTidy command, which a number of applications use.

Example

```
*RMFaster BASIC
```

Related commands

None

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*RMInsert

Reverses the action of a previous *Unplug command

Syntax

```
*RMInsert module_title [ROM_section]
```

Parameters

module_title	the title of any ROM resident module
ROM_section	ROM section to restrict command to

Use

*RMInsert reverses the action of a previous *Unplug command, but without reinitialising any modules.

If no ROM section number is specified, then this command clears the unplug bit for all versions of the specified module present in the machine.

If a ROM section number is specified, then this command clears the unplug bit for all versions of the specified module present in the given section. ROM section numbers are:

ROM section	Meaning
–1	System ROM
0	Expansion card 0
1	Expansion card 1
2	Expansion card 2
3	Expansion card 3
–2	Extension ROM 1
–3	Extension ROM 2
–4	Extension ROM 3 (etc)

This command is not available in RISC OS 2.

Example

```
*RMInsert MIDI 1
```

Related commands

*RMReInit, *Unplug

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*RMKill

Deactivates and deletes a relocatable module

Syntax

`*RMKill` *`module_title`*`[%`*`instantiation`*`]`

Parameters

`module_title`	the title of any currently installed module
`instantiation`	the instantiation of any currently installed module

Use

*RMKill deactivates the preferred instantiation of a relocatable module (or the specified instantiation if the second argument is used) and releases its workspace. If the module is in RAM, it is also deleted. If it is ROM resident, it is made inactive until reinitialised by the *RMReInit command, or until the next hard reset. Use this command only with extreme caution, as it may be drastic in its effects.

Example

```
*RMKill Debugger
```

Related commands

*RMClear, *RMInsert, *RMReInit, *RMTidy, *Unplug

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*RMLoad

Loads and initialises a relocatable module

Syntax

`*RMLoad` *`filename`* [*`module_init_string`*]

Parameters

`filename`	a valid pathname specifying a module file

Use

*RMLoad loads and initialises a relocatable module. It can then be accessed by the help system, and can provide SWIs and * Commands if available.

The file must have file type &FFA, otherwise the module handler will refuse to load it.

The optional initialisation string can be used to pass parameters to certain modules so they initialise themselves in a particular way. For example, you might use it to specify the amount of workspace that the module should claim, or a file that the module should load.

Example

```
*RMLoad WaveSynth $.Waves.Brass14
```

Related commands

*RMRun

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*RMReInit

Reinitialises a relocatable module

Syntax

```
*RMReInit module_title [module_init_string]
```

Parameters

module_title	the title of any currently installed module, active or otherwise
module_init_string	optional parameters to the module

Use

*RMReInit reinitialises a relocatable module, reversing the action of any previous *RMKill or *Unplug command. The module is returned to the state it was in when it was loaded. Use this command only with extreme caution, as it may be drastic in its effects.

- If the specified module is active, then it is killed and then re-initialised.
- If the specified module is not active, but is in the ROM, then the unplug bit in CMOS RAM is cleared for all versions of the specified module, and then the newest version of the module is initialised. (Under RISC OS 2 it is the first found version that is initialised.)

The optional initialisation string can be used to pass parameters to certain modules so they reinitialise themselves in a particular way. For example, you might use it to specify the amount of workspace that the module should claim, or a file that the module should load.

This command can produce unexpected results, for a variety of reasons. For example:

- The order of module initialisation is important in RISC OS. If a module relies on a second module being later initialised, you cannot successfully reinitialise the first module without then reinitialising the second.
- Under the desktop, a reinitialised module does not get restarted as a task unless you re-enter the desktop.

Example

```
*RMReInit Debugger
```

Related commands

*RMClear, *RMInsert, *RMKill, *RMTidy, *Unplug

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*RMRun

Loads and initialises a relocatable module, and then runs it

Syntax

`*RMRun` *`filename`* [*`module_init_string`*]

Parameters

`filename`	a valid pathname specifying a module file
`module_init_string`	optional parameters to the module

Use

*RMRun loads and initialises a relocatable module, and then runs it.

The module is first loaded and initialised. (This is equivalent to a call to *RMLoad.) The module can then be accessed by the help system, and can provide SWIs and * Commands if available.

The file must have file type &FFA, otherwise the module handler will refuse to load it.

The module is then run, if it can be. This is equivalent to an enter operation in OS_Module. Consequently, if the module cannot be run, then this command is equivalent to a *RMLoad command.

Example

```
*RMRun My_Module
```

Related commands

*RMLoad

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*RMTidy

Compacts the module area and reinitialises all the modules it contains

Syntax

```
*RMTidy
```

Parameters

None

Use

*RMTidy collects together free space in the module area by moving and reinitialising all the modules it contains. The free space is gathered into a consecutive chunk of memory.

Use this command only with extreme caution, as it is so drastic in its effects.

Related commands

*RMClear

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*ROMModules

Displays information about all relocatable modules currently installed in ROM

Syntax

```
*ROMModules
```

Parameters

None

Use

*ROMModules displays information about all relocatable modules which are currently installed in ROM.

The command displays the number allocated to each module, whether it is part of the system or in expansion cards or in an extension ROM, its name, and its status: active, running, dormant or unplugged. (Note that RISC OS 2 does not support extension ROMs, nor does it give a version number or report modules as running.)

- The names listed by this command are the module titles, which are used as parameters for other commands such as *RMKill.

System modules are stored in ROM, but may still be *RMKilled, *Unplugged, or replaced by RAM-based modules.

Example

```
*ROMModules
No. Position    Module name             Version Status
  1 System ROM  UtilityModule           2.20    Active
  2 System ROM  Podule                  1.23    Active
  3 System ROM  FileSwitch              1.98    Active
  4 System ROM  ResourceFS              0.09    Active
  5 System ROM  Messages                0.16    Active
        …
  1 Podule 1    Support16a              1.00    Active
        …
  1 Extn ROM 1  Tube6502Emulator        1.17    Dormant
  1 Extn ROM 2  Turbo6502Emulator       1.17    Dormant
  1 Extn ROM 3  Tube6502Emulator        1.17    Active
  2 Extn ROM 3  Turbo6502Emulator       1.17    Active
  1 Extn ROM 4  FontManager             2.85    Active
        …
```

or under RISC OS 2:

```
*ROMModules
No. Position   Module Name              Status
  1 System ROM UtilityModule            Active
  2 System ROM FileSwitch               Active
  3 System ROM Desktop                  Active
        …
  1 Podule 0   MailBleep                Dormant
  2 Podule 0   ROMBoard                 Dormant
        …
```

Related commands

*Modules

Related SWIs

OS_Module (page 1-228)

Related vectors

None

*Unplug

Kills and disables all copies of a ROM resident module

Syntax

*Unplug [*module_title* [*ROM_section*]]

*Unplug [*module_title*] (RISC OS 2)

Parameters

module_title	the title of any ROM resident module
ROM_section	ROM section to restrict command to – this parameter is not recognised by RISC OS 2

Use

*Unplug kills all copies of the named ROM module, releasing any workspace used. (In RISC OS 2 only the first copy found is deleted.) It also disables all versions of that module – whether in the system ROM, expansion cards or extension ROMs – by preventing them from being initialised (and hence available for use). This setting is stored in the CMOS RAM, and so is permanent even across a reset. To enable the module again you must use the *RMReInit or *RMInsert command. (The latter command is not available in RISC OS 2.)

If you supply a ROM section parameter, *Unplug restricts its effects to modules that are in that ROM section. ROM section numbers are:

ROM section	Meaning
–1	System ROM
0	Expansion card 0
1	Expansion card 1
2	Expansion card 2
3	Expansion card 3
–2	Extension ROM 1
–3	Extension ROM 2
–4	Extension ROM 3 (etc)

You should use this command with caution, otherwise you may find programs stop working because you have unplugged a module that is essential to them.

If no parameters are given, the unplugged ROM modules are listed.

Example

```
*Unplug RAMFSFiler
```
disables the RAMFSFiler module

Related commands

*RMInsert, *RMKill, *RMReInit

Related SWIs

OS_Module (page 1-228)

Related vectors

None

15 Program Environment

Introduction

The program environment refers to the conditions under which a program or module executes. There are three aspects to this environment.

- The memory used by the code and allocated for transient workspace.
- The handlers used by a program or module.
  A handler is a piece of code called when certain conditions occur. RISC OS provides a set of default handlers, so that a sensible default action will occur under such conditions. Here is a brief list of the kinds of conditions that we are talking about:
  - an error
  - an escape condition
  - an event
  - certain hardware exceptions, such as an undefined instruction
  - a break point
  - an unknown SWI being called
  - a program or module terminating.
- The system variables are a textual way of finding information about various aspects of the system. There are several kinds of variables:
  - string variables which contain characters only
  - integer variables which contain an integer
  - macro variables which are like string variables, except that they can contain references to special characters and other system variables.

Overview and Technical Details

Executing code

There are several ways of executing a piece of code. You can:

- *RMRun a module
- OS_Module ‘Enter’ a module
- *Run a program
- *Go, to execute a program in memory

Modules

The first two are described in the chapter entitled *Modules*. They are really the same thing. When a file is *RMRun, it is loaded into the relocatable module area. Its initialisation code is called, so that it can claim workspace etc, then its start code is called.

A module can also cause its own start entry point to be called if it wants to become the current application, using OS_Module. BASIC is an example of this. The *BASIC command is recognised by the OS using the BASIC module’s * Command table. The OS calls the routine which handles the *BASIC command, and this routine calls OS_Module with the reason code ‘enter’. For details on calling modules see the chapter entitled *Modules* on page 1-201.

Programs on file

The third case applies to files which have no file type, or have type &FF8. In the first case, the file is loaded at its load address, then it is started as an application through its execution address. If the file type is &FF8, the file is loaded at &8000 and started as an application there. See also the section entitled *Transient programs* below.

Programs in memory

Finally, if you call a machine code program using the *Go command, it becomes the current application. (This implies that you shouldn’t use *Go to call RAM-based routines from a language, as the routine can’t return – R14 contains no return address at this point.)

In all of these cases, the program is called in user mode, with interrupts enabled. Where a module is called, R12 points to the module’s private word.

Transient programs

A file with type &FFC (Utility) must contain position independent code. When such a file is *Run, it is loaded into the RMA and executed. This is used when you want to run a utility and then return to the program environment that you were in before running it. On entry to a transient program, registers are as follows:

R0 = pointer to command line
R1 = pointer to command tail
R12 = pointer to workspace
R13 = pointer to workspace end (stack)
R14 = return address
User mode, interrupts enabled

The workspace is 1024 bytes long, in the location given by R12 and R13 on entry. If more is required, it may be allocated from the RMA. The utility should return using MOV PC,R14 (freeing any extra workspace first). It does not become the current application and must not call OS_Exit; see the section entitled *Ending a task* below.

Note that R0 points to the first character of the command name, and R1 points to the first character of the command tail (with spaces skipped). This will be a control character if there were no parameters.

When a utility returns, the space it occupies is freed. Utilities are nestable – you can execute one utility from within another.

Note that utilities are viewed as system extensions. This means that they must only use the X form SWIs, so that the error handler is not called by their actions. A utility can return with an error by setting V and pointing R0 at an error block as usual.

Ending a task

Before describing the calls which control the application program's environment, it is worth explaining how to leave an application. In general, a simple 'return from subroutine' using MOV PC,R14 won't suffice. Instead, you should use a routine called OS_Exit (page 1-303). This passes control back to a well-defined place, which defaults to the supervisor * prompt, but could equally be a location in the previous application.

*Quit is equivalent to a call to OS_Exit.

OS_ExitAndDie (page 1-323) is like OS_Exit, but will kill a named module as well. This may be used, for example, when a module is specific to a particular application and you wish to kill the module when the application exits.

System variables

The system variables, maintained by the operating system in the system heap, provide a convenient way by which programs can communicate. Variables are accessed by their textual name. The name may contain any non-space, non-control character. When a variable is created, the case of the letters is preserved. However, when names are looked up the case is ignored, and you can use the characters '#' and '*' – just like looking up filenames.

Naming

You should avoid the use of wholly numeric names for system variables, such as 123, as this causes difficulties when the GS string operations are used to look up a variable's contents. In particular, they will always take <123> to mean the ASCII code 123, and will not attempt to look up the name as a variable. See the chapter entitled *Conversions* on page 1-453 for details of the GS calls, specifically OS_GSRead and OS_GSTrans.

Types

There are several types of system variable:

- String variables can contain any characters you like; these are returned when the string is read. They can be set with *Set.
- Integer variables are four-byte signed integers. They can be set with *SetEval.
- Macros are strings that are passed through OS_GSTrans when the string is read. This means that if the macro contains references to variables or other OS_GSReadable items, the appropriate translation takes place whenever the variable is accessed. They can be set with *SetMacro.

  A classic example of using a macro is to set the command line prompt CLI$Prompt to the current time using:

  ```
  *SetMacro CLI$prompt <Sys$Time><&20>
  ```

  Every time the prompt is displayed, it shows the current time, followed by a space.
- The final type of variable is machine code routines. A routine is called whenever the variable is to be read, and another when it is set. This allows great flexibility in the way in which such variables behave. For example, you could make a variable directly control a CMOS RAM location using this technique. Sys$Time is a good example of a code variable.

All the above types can be set with OS_SetVarVal (page 1-316) and read with OS_ReadVarVal (page 1-314).

Any non-code variable can be removed using *Unset. *Show will list the setting of one or more variables.

Miscellaneous environment features

OS_GetEnv (page 1-301) is a multi-purpose SWI that provides three useful pieces of information:

1 The address of the * Command string used to run the program.

This can be processed with OS_ReadArgs, which is described on page 1-478 of the chapter entitled *Conversions.*

2 The real time that the program was started.

3 The maximum amount of memory available to the program.

This can be altered with reason code 0 of OS_ChangeEnvironment; see page 1-320 for more details.

OS_WriteEnv (page 1-322) allows you to set the program start time and the command string.

Handlers

Handlers are short routines used to cope with special conditions that can occur under RISC OS. Here is a complete list of the handlers:

Handler
Undefined instruction
Prefetch abort
Data abort
Address exception
Error
CallBack
BreakPoint
Escape
Event
Exit
Unused SWI
UpCall

All of the calls that install user handlers pass through ChangeEnvironmentV. This can be intercepted to stop a subprogram changing parts of the environment that its parent wants to keep: for example, a debugger.

Before reading this section, you should be familiar with the chapters entitled *Software vectors* on page 1-63 and *Hardware vectors* on page 1-113, since many of these handlers are directly called from these vectors.

SWIs

OS_ChangeEnvironment (page 1-320) is the central SWI for handlers. There are several other routines that perform subsets of its actions. You are strongly recommended to use OS_ChangeEnvironment in any new applications as the others are only provided for compatibility.

The other calls are OS_Control (page 1-299), OS_SetEnv (page 1-305), OS_CallBack (page 1-307), OS_BreakCtrl (page 1-310) and OS_UnusedSWI (page 1-312).

OS_ReadDefaultHandler allows you to get the address and details of any of the default handlers. This would be used if you wished to set up a well-defined state before running a subprogram: for example, the Desktop does so.

Details of Handlers

When a handler is called, you should not expect to be able to see the foreground application's registers. You should **only** rely on those registers explicitly defined in each handler as being meaningful on entry.

You should take care not to corrupt R14_SVC during handler code. This implies saving it on the stack if you use SWIs; see the chapter entitled *Interrupts and handling them* on page 1-119 for details. The details of each of the handlers follows:

Undefined instruction, Prefetch abort, Data abort and Address exception

These handlers are all called from hardware vectors. For a description of them see the chapter entitled *Hardware vectors* on page 1-113. These handlers are all entered with the processor in SVC mode.

All of the default handlers simply generate errors, which are passed to the current error handler.

Error

The error handler is called after any error has been generated. It is called by the default routine on the error vector; thus any routines using this vector should always 'pass it on'. Continuing after an error is not generally recommended. You should always use the X form SWIs if you wish to stay in control even when an error occurs.

The error handler is entered in User mode with interrupts enabled. Note that if the error handler is set up using OS_ChangeEnvironment, the workspace pointer is passed in R0, not R12 as is usual for other handlers.

The error handler must provide an error buffer of size 256 bytes, the address of which should be set along with the handler address. On an error the buffer will be set to contain the following:

Offset	Contents
0 - 3	PC when error occurred
4 - 7	Error number provided with the error.
8…	Error string, terminated with a 0

The default error handler reports the error message and number – although applications frequently set up their own error handlers. BASIC is one such example.

BreakPoint

This handler is called when the SWI OS_BreakPt (page 1-309) is called. All the user mode registers are dumped into a buffer (the *register save block*), and then the handler is entered in SVC mode.

When setting the address of a replacement break point handler you must also specify the address of the register save block, which must be word aligned and 16 words long. You can also specify a pointer to workspace to pass in R12 when your handler is called.

The following code is suitable to restore the user registers and return:

```
ADR     R14, saveblock          get address of saved registers
LDMIA   R14, {R0-R14}^          load user registers from block;
                                note that user R13,R14 are altered
MOV     R0, R0                  no-op after forcing User mode
LDR     R14, [R14,#15*4];       load user PC into SVC R14
MOVS    PC, R14                 return to correct address and mode
```

The default handler displays the message 'Break point at &*xxxxx*' and calls OS_Exit.

Escape

This handler is called when an escape condition is detected. See the chapter entitled *Character Input* on page 1-863 for details of this. You can specify a pointer to workspace to pass in R12 when this handler is called.

When the handler is entered, registers have the following values:

R11	bit 6 set, implying escape condition
R12	pointer to workspace, if set up – should never be 1
R13	a full, descending stack pointer

To continue after an escape, the handler should reload the PC with the contents of R14. If R12 contains 1 on return then the CallBack flag is set; for details of the action this causes, see the section entitled *CallBack* on page 1-295. Typically (eg for BASIC), the handler will set an internal flag which is checked by the foreground program.

Event

This handler is called by the default owner of EventV when an event occurs. You can specify a pointer to workspace to pass in R12 when this handler is called.

When the handler is entered the processor is in either SVC or IRQ mode, with the following register values:

R0	event reason code
R1…	parameters according to event code
R12	pointer to workspace, if set up – should never be 1
R13	a full, descending stack pointer

To continue after an event, the handler should reload the PC with the contents of R14. Again, if R12 contains 1 on return then the CallBack flag is set; for details of the action this causes, see the section entitled *CallBack* on page 1-295.

Exit

This handler is called when the SWIs OS_Exit (page 1-303) or OS_ExitAndDie (page 1-323) are called. It is entered with the processor in user mode. You can specify a pointer to workspace to pass in R12 when this handler is called.

Unused SWI

This handler is called by the default owner of the UKSWIV. (If RISC OS can't decode the number of a SWI into one which it supports directly, it offers it as a service call to modules. If none of them claim the service, it then calls the vector UKSWIV. This allows a user routine on that vector to try to deal with the SWI. If there is no such routine, or the one(s) that is present passes the call on, then the default owner of the vector calls the Unused SWI handler.)

You can specify a pointer to workspace to pass in R12 when this handler is called.

When the handler is entered the processor is in SVC mode, with interrupts in the same state as the caller. The registers have the following values:

R11	SWI number (Bit 17 clear)
R13	SVC stack pointer
R14	user PC with V cleared

R10, R11 and R12 are stacked and are free for your own use.

UpCall

This handler is called by the default owner of UpCallV when OS_UpCall (page 1-190) is called. OS_UpCall is used to warn your program of errors and situations that you may be able to recover from. See the chapters entitled *Software vectors* on page 1-63 and *Communications within RISC OS* on page 1-179. You can specify a pointer to workspace to pass in R12 when this handler is called.

CallBack

This handler is called whenever RISC OS's internal CallBack flag is set, and the system next exits to User mode with interrupts enabled. It uses a register save block (the address of which should be set along with the handler address) in which all the registers are dumped when the handler is called. This must be word-aligned and 16 words long. You can specify a pointer to workspace to pass in R12 when this handler is called. A more detailed description follows.

CallBacks in more detail

There are two types of CallBack usage under RISC OS:

- Transient CallBacks are placed in a list by calling OS_AddCallBack (page 1-324). They are used to deal with a specific case, and are called once before being removed.
- The CallBack handler is permanent and takes all CallBacks that are not intercepted by transients. These CallBacks are explicitly requested by calling OS_SetCallBack (page 1-313). They can also be implicitly requested by setting R12 to 1 on exit from either an escape or event handler. There is a system default CallBack handler, but you can of course replace it using OS_ChangeEnvironment.

Transient CallBacks

Transient CallBacks may be called on the system being threaded out of – that is, when it enters User mode with interrupts enabled. They can also be called when RISC OS is idling; for example, while it is waiting in OS_ReadC.

Transient CallBacks are usually set up by an interrupt routine that needs to do complex processing that would take too long in an interrupt, or that needs to call a non-re-entrant SWI. OS_AddCallBack tells RISC OS that the interrupt routine wishes to be 'called back' when the machine is in a state that no longer imposes the restrictions associated with an interrupt routine. OS_RemoveCallBack removes a transient CallBack; this is most useful if the module is being killed before the transient CallBack has been serviced.

Transient CallBacks can safely be used by many clients.

Other CallBacks

The CallBack handler is only ever called on the system being threaded out of – that is, when it enters User mode with interrupts enabled. Unlike transient CallBacks, it is not called when RISC OS is idle. This means that you cannot rely on being called back within **any** given time. You **must** take this into consideration before using a CallBack handler.

Also, you **must not** allow a second CallBack before your first one has completed; see the section entitled *Application Notes* on page 1-338 for an example of how to implement a semaphore to prevent this.

The CallBack code is called in IRQ or supervisor mode with interrupts disabled. The PC stored in the save block will be a user mode PC with interrupts enabled. Note that if the currently active program has interrupts disabled or is running in supervisor mode, CallBack is not used.

In the simple case the CallBack routine should be exited by:

```
ADR     R14, saveblock          get address of saved registers
LDMIA   R14, {R0-R14}^          load user registers from block –
                                note that user R13,R14 are altered
MOV     R0, R0                  no-op after forcing User mode
LDR     R14, [R14,#15*4];       load user PC into SVC R14
MOVS    PC, R14                 return to correct address and mode
```

In RISC OS 3 (version 3.10) or later, the supervisor stack must also be empty when the CallBack handler is called. This ensures that certain module SWIs that temporarily enter User mode (so that transient CallBacks are called) do not cause the CallBack handler to be called.

Currently active object pointer

This is a pointer to the address of: the last application started, or the last error handler called, or the last exit handler called. It is used by OS_Module to determine whether a module can be killed.

Setting up and restoring the environment

In order to deal correctly with the various ways in which applications can be run, and killed off, the following approach has been developed for setting up the program environment when an application starts, and restoring it when it is killed. The basic problems are:

- if a new application is started ‘on top’ of the currently active one, it should completely replace the first, and should therefore have the same ‘parent’ environment as the first application.
- if the currently active application is killed off, it must restore its ‘parent’ environment.

Using high level languages

Typically these are handled for you by run-time language libraries, and so if you are writing in a high-level language you do not need to worry.

Using machine code or writing a run-time language library

However, if you are yourself writing a run-time language library, or if you are writing your application in machine code (for example as a module which runs as a Wimp task) you must take one of these two possible approaches:

- Do not set up **any** handlers **at all**, and **always** call the ‘X’ form of SWIs, to avoid calling the error handler. If the error handler is called, the application will be terminated, as the parent error handler will be invoked.

- Set up Error, Exit and UpCall handlers **as described below**, so that the program environment can be restored correctly when the program terminates. You **must** provide all three of these handlers if you use any handlers at all, otherwise there will be some circumstances in which your application can be replaced or killed without restoring its 'parent' environment.

Starting an application

When you start an application, you must:

1 Check that there is sufficient memory to do so – if not, call OS_GenerateError ('Not enough application memory')

2 Set up your handlers using the SWI XOS_ChangeEnvironment; store the values returned in R1-R3 so you can later restore the old handlers.

Note that you **must** store the previous values not only for Exit, Error and UpCall handlers, but also **for any other handlers that are set up**.

If your Error handler is called

If your error handler is called and you want to call the 'external' error handler (eg BASIC if '–quit' was on the command line), you should:

1 restore **all** handlers to their original values (R1 - R3 for each)

2 call OS_GenerateError.

If your Exit handler is called

If your exit handler is called you should:

1 restore **all** handlers to their original values (R1 - R3 for each)

2 call OS_Exit.

If your UpCall handler is called

If your UpCall handler is called and R0 = UpCall_NewApplication (256), you should:

1 restore **all** handlers to their original values (R1 - R3 for each)

2 return to the caller, preserving all registers (ie carry on and start the new application).

Summary

The approach described above ensures that it is not possible for your application to be terminated without it first restoring all handlers to their original values.

SWI Calls

OS_Control (SWI &0F)

Read/write handler addresses

On entry

R0 = pointer to error handler, or 0 to read
R1 = pointer to error buffer, or 0 to read
R2 = pointer to escape handler, or 0 to read
R3 = pointer to event handler, or 0 to read

On exit

R0 = pointer to previous error handler
R1 = pointer to previous error buffer
R2 = pointer to previous escape handler
R3 = pointer to previous event handler

Interrupts

Interrupts are not enabled
Fast interrupts are enabled

Processor Mode

Processor is in IRQ or SVC mode

Re-entrancy

SWI cannot be re-entered as interrupts are disabled

Use

OS_Control sets some of the exception handlers. The addresses of the error handler, error handler buffer, escape handler and event handler are passed in R0 - R3. Zero for any of these means no change – hence you can read the current value. The error buffer must be 256 bytes long.

Note that the call OS_ChangeEnvironment provides all of the facilities that this call provides, and **should be used in preference**. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_ChangeEnvironment (page 1-320)

Related vectors

ChangeEnvironmentV

OS_GetEnv (SWI &10)

Read environment parameters

On entry

—

On exit

R0 = pointer to environment string
R1 = permitted RAM limit (ie highest address available + 1)
R2 = pointer to real time the program was started (5 bytes)

Interrupts

Interrupt status is unaltered
Fast interrupt status is unaltered

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI reads some information about the program environment.

The environment string pointed to by R0 is normally a copy of the command line used to start the program. However, there may be some circumstances where a command line was not used to start the program; in such cases a meaningful string is still passed. For example, if a module is started using OS_Module 2, the string passed will be '<module title> <parameter string passed in R2 on entry to OS_Module>'.

R1 returns the address of the byte above the last one available to the application. You can alter this using reason code 0 of OS_ChangeEnvironment.

The five bytes pointed to by R2 give the real time the program was started: ie centiseconds since 00:00:00 01–Jan–1900.

You can set these values using OS_WriteEnv.

Related SWIs

OS_ChangeEnvironment (page 1-320), OS_WriteEnv (page 1-322)

Related vectors

None

OS_Exit (SWI &11)

Pass control to the most recent exit handler

On entry

R0 = pointer to error buffer
R1 = 'ABEX' (&58454241) if return code is to be set
R2 = return code

On exit

Never returns

Interrupts

Interrupt status is unaltered
Fast interrupt status is unaltered

Processor Mode

Processor is in USR mode

Re-entrancy

SWI is not re-entrant

Use

When OS_Exit is called, control returns to the most recent exit handler. The BASIC statement QUIT performs an OS_Exit. Before executing OS_Exit, however, you must restore any of the handlers changed in starting the application.

If the exiting program wishes to return with a return code, it must set R1 to the hex value shown above, and R2 to the desired value. The value should be zero to indicate no error; otherwise the value should indicate the severity of the error, so 1, for example might indicate a trivial error or warning. The return value is assigned to the variable Sys$ReturnCode, which can be interrogated by any program using OS_ReadVarVal.

If the returned value is greater than the value of the system variable Sys$RCLimit, RISC OS also gives the error 'Return code limit exceeded' (&1E2). The user can alter the value of Sys$RCLimit to control which errors are returned; your application should not itself alter the variable.

Related SWIs

OS_ExitAndDie (page 1-323)

Related vectors

None

OS_SetEnv (SWI &12)

Set environment parameters

On entry

R0 = pointer to exit handler, or 0 to read
R1 = permitted RAM limit (ie highest address available + 1), or 0 to read
R4 = pointer to undefined instruction handler, or 0 to read
R5 = pointer to prefetch abort handler, or 0 to read
R6 = pointer to data abort handler, or 0 to read
R7 = pointer to address exception handler, or 0 to read

On exit

R0 = pointer to previous exit handler
R1 = previous permitted RAM limit (ie highest address available + 1)
R4 = pointer to previous undefined instruction handler
R5 = pointer to previous prefetch abort handler
R6 = pointer to previous data abort handler
R7 = pointer to previous address exception handler

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_SetEnv sets several of the handlers for a program.

Note that the call OS_ChangeEnvironment provides all of the facilities that this call provides, and **should be used in preference**. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_ChangeEnvironment (page 1-320)

Related vectors

ChangeEnvironmentV

OS_CallBack (SWI &15)

Set up the CallBack handler

On entry

R0 = pointer to CallBack register save block, or 0 to read
R1 = pointer to CallBack handler, or 0 to read

On exit

R0 = pointer to previous CallBack register save block
R1 = pointer to previous CallBack handler

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_CallBack sets up the address of the CallBack handler and the register save block, zero for either value meaning no change – hence you can read the current value. The register save block must be word-aligned and 16 words long.

Note that the call OS_ChangeEnvironment provides all of the facilities that this call provides, and **should be used in preference**. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_ChangeEnvironment (page 1-320)

Related vectors

ChangeEnvironmentV

OS_BreakPt (SWI &17)

Cause a break point trap to occur and the BreakPoint handler to be entered

On entry

—

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

When OS_BreakPt is executed, all the user mode registers are saved in a block and the BreakPoint handler is called. The saved registers are only guaranteed to be correct for user mode.

The default handler displays the message ‘Break point at &*xxxxx*’ and calls OS_Exit.

This SWI would be placed in code by the debugger at required breakpoints.

Related SWIs

OS_BreakCtrl (page 1-310)

Related vectors

None

OS_BreakCtrl (SWI &18)

Set up the BreakPoint handler

On entry

R0 = pointer to BreakPoint register save block, or 0 to read
R1 = pointer to BreakPoint handler, or 0 to read

On exit

R0 = pointer to previous BreakPoint register save block
R1 = pointer to previous BreakPoint handler

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_BreakCtrl sets up the address of the BreakPoint handler and the BreakPoint register save block, zero for either value meaning no change – hence you can read the current value. The register save block must be word-aligned and 16 words long.

Note that the call OS_ChangeEnvironment provides all of the facilities that this call provides, and **should be used in preference**. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_BreakPt (page 1-309), OS_ChangeEnvironment (page 1-320)

Related vectors

ChangeEnvironmentV

OS_UnusedSWI (SWI &19)

Set up the handler for unused SWIs

On entry

R0 = pointer to unused SWI handler; or 0 to read

On exit

R0 = pointer to previous unused SWI handler

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not enabled

Use

OS_UnusedSWI sets up the address of the unused SWI handler, zero meaning no change – hence you can read the current value.

Note that the call OS_ChangeEnvironment provides all of the facilities that this call provides, and **should be used in preference**. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_ChangeEnvironment (page 1-320)

Related vectors

ChangeEnvironmentV

OS_SetCallBack (SWI &1B)

Cause a call to the CallBack handler

On entry

—

On exit

—

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered because interrupts are disabled

Use

OS_SetCallBack sets the CallBack flag and so causes entry to the CallBack handler when the system next exits to user mode code with interrupts enabled (apart, of course, from the exit from this SWI). This SWI may be used if the code linked into the system (via a vector or as a SWI handler, etc) is required to do things on exit from the system.

Related SWIs

OS_CallBack (page 1-307)

Related vectors

None

OS_ReadVarVal (SWI &23)

Read a variable value

On entry

R0 = pointer to variable name, which may be wildcarded (using '*' and '#')
R1 = pointer to buffer to hold variable value
R2 = maximum length of buffer, or bit 31 set to check existence/length of variable
R3 = context pointer (used with wildcarded names), or 0 for first call
R4 = 3 if an expanded string is to be converted on return

On exit

R0, R1 preserved
R2 = number of bytes read
R3 = new context pointer (null-terminated)
R4 = variable type

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadVarVal reads a variable and returns its value and its type.

Before reading a variable you must check the length of the data that will be returned. To do so, call XOS_ReadVarVal with R2 set to a value less than zero (bit 31 set) on entry. You can also use this to check for the existence of a variable.

- If the variable exists, R2 will still be negative on exit; furthermore, if R4 ≠ 3 on entry (ie the variable is not an expanded string) the value of R2 is NOT (length of value).

- If the variable does not exist R2 will be zero on exit.

When using the call in this manner, you may get an error on exit, which you should ignore. This feature is not available under RISC OS 2; in this case you may assume that the length of the variable will be at most 256 bytes.

For a wildcarded name R3 should be 0 on entry the first time the call is made, and thereafter preserved from the previous call. On exit, R3 points to the name of the variable found. This enables all matches to be found. The XOS_ReadVarVal form of the call should be used if you don't want an error to occur after the last name has been found.

R4, if set to 3 on entry, indicates that a suitable conversion to a string should be performed. String variables are unaltered, numbers are converted to (signed) decimal strings, and macros are OS_GSTrans'd.

If R4 isn't 3 on entry, the un-OS_GSTrans'd version of a macro is returned, and the four-byte binary of a number is returned.

The type of the variable read is returned in R4 as follows:

Value		Type
VarType_String	(0)	String
VarType_Number	(1)	4 byte (signed) integer
VarType_Macro	(2)	Macro

Returned strings are not terminated, and you should use the length returned in R2 when reading them.

See the section entitled *Application Notes* on page 1-338 for an example of reading a variable.

Related SWIs

OS_SetVarVal (page 1-316)

Related vectors

None

OS_SetVarVal (SWI &24)

Write a variable value

On entry

R0 = pointer to variable name, which may be wildcarded (* and #) if updating/deleting
R1 = pointer to variable value
R2 = length of value, or negative to delete the variable
R3 = context pointer (used with wildcarded names), or 0 for first call
R4 = variable type

On exit

R0 - R2 preserved
R3 = new context pointer (null-terminated)
R4 = variable type created if expression is evaluated

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_SetVarVal either creates, updates or deletes a variable. The variable's name is pointed to by R0, and may be terminated by any character whose ASCII value is 32 or less. It may be wildcarded if the variable is to be updated or deleted (ie if it already exists).

To delete a variable, R2 must be negative on entry; if the variable is a code variable (see below), the variable type in R4 **must** be set (ie R4 = 16).

When creating or updating a variable, R1 must point to the value to be assigned. The interpretation of this value depends on the type given in R4 as follows:

Value	Type	
VarType_String	(0)	Value is a string which will be OS_GSTrans'd immediately
VarType_Number	(1)	Value is a 4 byte (signed) integer
VarType_Macro	(2)	Value is a string which will be OS_GSTrans'd each time it is used
VarType_Expanded	(3)	Value is a string which will be evaluated as an expression using OS_EvaluateExpression, and assigned to a number or string variable, depending on the expression type
VarType_LiteralString	(4)	Value is a literal string (ie it will not be OS_GSTrans'd)
VarType_Code	(16)	Special case (see below)

With the exception of a literal string, all strings must be terminated by a linefeed (ASCII 10) or carriage return (ASCII 13) or null (ASCII 0).

If the call is successful, R3 is updated to point to the new context so allowing the next match of a wildcarded name to be obtained on a subsequent call. R4 returns the type created if an expression was evaluated (ie if R4 was 3 on entry).

VarType_Code

When R4 is set to 16 on entry (and R2 ≥ 0) a code variable may be created. In this case R1 is the pointer to the code fragment associated with the variable, and R2 is the length of the code fragment. This code must be word-aligned and takes the following format:

Offset	Contents
0	Branch instruction to entry point for write operation
4	Entry point for read operation
8…	Body of code…

Values are always written to (and read from) code variables as strings. The entry for the write operation is called whenever the variable is to be set, as follows:

On entry

R1 = pointer to the value to be used
R2 = length of value

On exit

R1, R2, R4, R10 - R12 may be corrupted

The entry for the read operation is called whenever the variable is to be read by a call to OS_ReadVarVal, as follows:

On entry

—

On exit

R0 = pointer to value
R1 = corrupted
R2 = length of value

Both entries are called in SVC mode, therefore if any SWIs are used, R14 must be saved on the stack so that it does not become corrupted. The SVC stack is used, and no workspace is reserved. You can return errors by setting the V flag as usual.

See the section entitled *Application Notes* on page 1-338 for an example of a code variable.

Note that when a function key is input, the appropriate variable Key$*n* is read using OS_ReadVarVal. Therefore by creating your own code variables with these names, you can cause the reading of a function key to cause a routine to be called instead of just a string being read.

Errors

OS_SetVarVal can return the following errors:

- Bad name — Wildcards/control characters in name when creating
- Bad string — OS_GSTrans unable to translate string
- Bad macro value — Control characters in the value string (R1)
- Bad expression — Expression cannot be evaluated
- Variable not found — For deletion or update
- No room for variable — Not enough room to create/update it (system heap full)
- Variable value too long — Variables are limited to 256 bytes in RISC OS 2 and RISC OS 3 (version 3.00)
- Bad variable type

Related SWIs

OS_ReadVarVal (page 1-314)

Related vectors

None

OS_ChangeEnvironment (SWI &40)

Install a handler

On entry

R0 = handler number
R1 = pointer to new handler, or 0 to read
R2 = value of R12 with which to call the handler, or 0 to read
R3 = pointer to buffer (if appropriate), or 0 to read

On exit

R0 preserved
R1 = pointer to previous handler
R2 = previous value of R12 with which to call the handler
R3 = pointer to previous buffer

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ChangeEnvironment is a single routine which performs the actions of OS_Control, OS_SetEnv, OS_CallBack, OS_BreakCtrl, and OS_UnusedSWI. In fact, all of those routines use this call. In new programs, you should always use this call in preference to the earlier ones.

For full details of the handlers, see the section earlier in this chapter.

On entry, R0 contains a code which determines which particular handler's address is to be set up. The new address is passed in R1. R0 also determines whether R2 and R3 are relevant or not. This is summarised in the table below:

R0	Handler	R2	R3
0	MemoryLimit	Ignored	Ignored
1	Undefined instruction	Ignored	Ignored
2	Prefetch abort	Ignored	Ignored
3	Data abort	Ignored	Ignored
4	Address exception	Ignored	Ignored
5	Other exceptions	Ignored	Ignored
6	Error	R0 when called	Error buffer address
7	CallBack	R12 when called	Register buffer address
8	BreakPoint	R12 when called	Register buffer address
9	Escape	R12 when called	Ignored
10	Event	R12 when called	Ignored
11	Exit	R12 when called	Ignored
12	Unused SWI	R12 when called	Ignored
13	Exception registers	Ignored	Ignored
14	Application space	Ignored	Ignored
15	Currently active object	Ignored	Ignored
16	UpCall	R12 when called	Ignored

The ‘Memory limit’ (handler 0) is the permitted RAM limit, as used by OS_GetEnv. The ‘Application space’ (handler 14) is the amount of read/write memory in application space. Consequently it should always be the case that Application space ≥ Memory limit.

‘Other exceptions’ (handler 5) is for future expansion.

The error buffer (handler 6) must be 256 bytes long.

The register buffers (handlers 7 and 8) must be word-aligned and 16 words long.

Handler 13 sets the address of the area in memory where the registers are dumped when one of the exceptions (1 - 5) occurs, if the default handlers are used. Again, this must be word-aligned and 16 words long.

Note that in order to perform its function, OS_ChangeEnvironment vectors through ChangeEnvironmentV. A routine linked onto this vector can stop the change from happening by setting R1 (and if appropriate R2, R3) to zero and passing the call on; see the chapter entitled *Software vectors* on page 1-63**.**

Related SWIs

OS_Control (page 1-299), OS_SetEnv (page 1-305), OS_CallBack (page 1-307) OS_BreakCtrl (page 1-310), OS_UnusedSWI (page 1-312)

Related vectors

ChangeEnvironmentV

OS_WriteEnv (SWI &48)

Set the program environment command string and start time

On entry

R0 = pointer to environment string
R1 = pointer to real time the program was started (5 bytes)

On exit

R0, R1 preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call sets the environment string and start time for an application, as returned by OS_GetEnv. For more details see OS_GetEnv on page 1-301.

This SWI is mainly used by debuggers.

Related SWIs

OS_GetEnv (page 1-301)

Related vectors

None

OS_ExitAndDie (SWI &50)

Kill a module and pass control to the most recent exit handler

On entry

R0 = pointer to error buffer
R1 = 'ABEX' (&58454241) if return code is to be set
R2 = return code
R3 = pointer to module name

On exit

never returns

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 SWI is like OS_Exit, except that it will kill a module before exiting. R3 points to a string containing the module's name. For more details, see OS_Exit on page 1-303.

Related SWIs

OS_Exit (page 1-303)

Related vectors

None

OS_AddCallBack (SWI &54)

Add a transient CallBack to the list

On entry

R0 = address to call
R1 = value of R12 to be called with

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call places a transient CallBack on a list of tasks who want to be called as soon as RISC OS is not busy. Usually, this will be just before returning from a SWI or while waiting for a key and so on.

This SWI is usually called from an interrupt routine that needs to do complex processing that would take too long in an interrupt, or that needs to call a non-re-entrant SWI. Note that you don't also need to call OS_SetCallBack, which is only needed when using the CallBack handler.

A routine called by this mechanism must preserve all registers and return by

```
MOV PC, R14
```

Related SWIs

OS_RemoveCallBack (page 1-327)

Related vectors

None

OS_ReadDefaultHandler (SWI &55)

Get the address of the default handler

On entry

R0 = reason code (0 - 16)

On exit

R0 preserved
R1 = pointer to default handler
R2 = pointer to workspace
R3 = pointer to buffer

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

Using the same handler number in R0 as those in OS_ChangeEnvironment (see page 1-321), this SWI returns details about the default handler.

Zero in R1, R2 or R3 on exit means that it is not relevant.

Related SWIs

OS_ChangeEnvironment (page 1-320)

Related vectors

None

OS_RemoveCallBack (SWI &5F)

Removes a transient CallBack from the list

On entry

R0 = address that was to be called
R1 = value of R12 that the routine was to be called with

On exit

R0, R1 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call removes a transient CallBack from the list. You should do so if your module has an outstanding CallBack request, but will not be able to service the request when it is granted – for example if the module is being killed.

This call is not available in RISC OS 2, which can cause problems. For example, if a module is being killed and it has outstanding CallBack requests, it must refuse to die, otherwise the CallBack may be granted after that memory has been reused for something else.

Related SWIs

OS_AddCallBack (page 1-324)

Related vectors

None

* Commands

*Go

Calls machine code at the given address

Syntax

```
*Go [hexadecimal_address] [ ; environment]
```

Parameters

hexadecimal_address	address of machine code to call
environment	environment string to pass to machine code

Use

*Go calls machine code at the given address, passing it an optional environment string. If the address is omitted, it defaults to &8000, which is where application programs (such as the C compiler) are loaded.

*Go enters an application, and you cannot use it to run machine code subroutines.

Example

`*Go 9000 ; SrcList`	Call machine code at &9000, passing it the string 'SrcList'

Related commands

None

Related SWIs

None

Related vectors

None

*Quit

Exits from the current application

Syntax

```
*Quit
```

Parameters

None

Use

*Quit exits from the current application – that is, it returns to the previous context.

Related commands

*GOS

Related SWIs

OS_Exit (page 1-303)

Related vectors

None

*Set

Assigns a string value to a system variable

Syntax

```
*Set varname value
```

Parameters

varname	a variable name, or a wildcard specification for a single variable name
value	string value to *GSTrans* and then assign to the system variable *varname*

Use

*Set assigns a string value to a system variable, like an assignment statement in a programming language. For example:

```
*Set varname text
```

assigns the string ‘text’ to the variable `varname`. The string is *OS_GSTrans’d* before it is assigned.

Aliases

Another use for the *Set command is to change the name of a command to one which is more convenient for the user:

```
*Set Alias$name cname
```

establishes *name* as an alternative name for the command *cname*; for example after:

```
*Set Alias$Aid Help
```

the command *Aid is now a synonym for *Help; both commands access the help system. Another example is:

```
*Set Alias$Mode Echo |<22>|<%0>
*Mode 12
```

The command implements a new command *Mode, which sets the screen to mode 12 (in the above case). The Echo command reflects the string which follows it; |<22> generates the ASCII character 22, Ctrl V, which is equivalent to the VDU command to change mode. |<%0> reads the first parameter from the command line, and generates the corresponding ASCII code.

The command `*Show Alias$*` lists all aliases.

Example

```
*Set Sys$Year 1992
```

Related commands

*SetEval, *SetMacro, *Unset

Related SWIs

OS_SetVarVal (page 1-316), OS_GSTrans (page 1-466)

Related vectors

None

*SetEval

Evaluates an expression and assigns its value to a system variable

Syntax

`*SetEval` *`varname expression`*

Parameters

`varname`	a valid variable name
`expression`	a valid Command Line expression

Use

*SetEval evaluates an expression and assigns its value to a system variable.

See the section entitled *Evaluation operators* on page 1-456 for a description of the operators that you can use.

Example

```
*Set rate 12
*SetEval rate rate + 1
*Show rate
rate (Number) : 13
```

```
*SetEval fred "jim"+"sheila"
*Show Fred
fred : jimsheila
```

Related commands

*Set, *SetMacro, *Unset, *Eval

Related SWIs

OS_SetVarVal (page 1-316)

Related vectors

None

*SetMacro

Assigns a string macro to a system variable

Syntax

`*SetMacro` *`varname macro`*

Parameters

`varname`	a variable name, or a wildcard specification for a single variable name
`macro`	string value to assign to the system variable *`varname`*, which is *GSTrans'd* each time it is read

Use

*SetMacro assigns a string value to a system variable, like function definition in a programming language. For example:

```
*Set varname text
```

assigns the string 'text' to the variable `varname`. The string is not *OS_GSTrans'd* before it is assigned; instead it is *OS_GSTrans'd* each time the variable is read.

Example

```
*SetMacro CLI$Prompt "<Sys$Time> "
13:43:17                    system time replaces existing prompt
Return                      Return key pressed two seconds later
13:43:19                    new system time displayed as prompt
```

This resets the Command Line prompt, which appears as the first item on each line, to be the current time whenever the prompt is given. Compare this with using the `*Set` command:

```
*Set fred <Sys$Time>
*Show fred
FRED  :  13:43:59
```

the `*Show` command issued five minutes later will still produce:

```
*Show fred
FRED  :  13:43:59
```

Notice that the time is fixed at the time the `*Set` command was performed, in contrast to the `*SetMacro` command.

Related commands

*Set, *SetEval, *Unset

Related SWIs

OS_SetVarVal (page 1-316), OS_GSTrans (page 1-466)

Related vectors

None

*Show

Displays the list of system variables

Syntax

`*Show [`*`variable_spec`*`]`

Parameters

`variable_spec`	a variable name or a wildcard specification for a set of variable names

Use

*Show displays the name, type and current value of any system variables matching the name given as a parameter. These include the ‘special’ system variables, which may be altered, but which cannot be deleted.

If no name is given, all system variables are displayed.

Example

`*Show`	*lists all system variables*
`*Show CLI$Prompt`	
`*Show Alias$*`	*lists all aliases*

Related commands

*Set, *SetEval, *SetMacro

Related SWIs

OS_ReadVarVal (page 1-314)

Related vectors

None

*Unset

Deletes a system variable

Syntax

`*Unset` *`variable_spec`*

Parameters

`variable_spec`	a variable name or a wildcard specification for a variable name

Use

*Unset deletes a system variable, which may be specified using wildcards.

Example

```
*Unset My_var
```

Related commands

*Set, *SetEval, *SetMacro

Related SWIs

OS_SetVarVal (page 1-316)

Related vectors

None

Application Notes

Reading a variable

Here is a short example of reading a variable using OS_ReadVarVal:

```
;Print all sys$ variable names
        ADR     R1, valBuffer           ;Buffer to place value
        MOV     R3, #0                  ;Initial context
.loop
        ADR     R0, strName             ;Wildcarded name to find
        MOV     R2, #bufferLen          ;Length of value buffer
        SWI     "XOS_ReadVarVal"        ;Non-error reporting one
        MOVVSS  PC,R14                  ;Return and clear V
        MOV     R0, R3                  ;Get address of name
        SWI     "OS_Write0"             ;Print it
        SWI     "OS_NewLine"            ;and new line
        B       loop                    ;again
        ....
.strName        EQUS    "SYS$*" + CHR$0
```

Checking the value of a variable

The short code fragment below checks if a variable has a particular value, without giving an error if it does not exist or contains quotes.

```
*SetMacro App$Temp <Variable>
If App$Temp = "desired_value" Then commands…
```

Don't forget to `*Unset` the *App*`$Temp` macro when you have finished using it.

Code variable

Below is a complete example of a program to create a variable called Mode. The read action is to return the current display mode, and the write action to set the mode.

```
.start  ADR     R0, varName             ;Pointer to the name
        ADR     R1, code                ;Start of code body
        MOV     R2, #endCode-code       ;Length of code body
        MOV     R3, #0                  ;Context pointer
        MOV     R4, #&10                ;'special' type
        SWI     "OS_SetVarVal"          ;Create it
        MOV     PC, R14                 ;Return

.code
        B       writeCode               ;Branch to write code
.readCode
        STMFD   R13!, {R14}             ;Save return address
        MOV     R0, #&87                ;OS_Byte read mode number
        SWI     "XOS_Byte"
        MOV     R0, R2                  ;Mode in R0 for conversion
        ADR     R1, buffer              ;Buffer for ASCII conversion
        MOV     R2, #4                  ;Max len of buffer
        SWI     "XOS_BinaryToDecimal"
        MOV     R0, R1                  ;Pointer in R0
                                        ;length already in R2
        LDMFD   R13!, {PC}              ;Return

.writeCode
        STMFD   R13!, {R0,R14}          ;Save registers
        SWI     "XOS_ReadUnsigned"      ;R1 set correctly already
        SWIVC   &20100+22               ;VDU mode change
        MOVVC   R0,R2                   ;Get integer read in R0
        SWIVC   "XOS_WriteC"            ;Do mode change
        LDMVCFD R13!, {R0,PC}           ;Return without error
        ADD     R13, R13, #4            ;Move stack pointer past R0, so we
        LDMFD   R13!, {R1,PC}           ;don't overwrite error pointer

.buffer
        EQUD    0                       ;Buffer for string conversion
.endCode

.varName
        EQUS    "Mode "                 ;Name of variable
```

The routine at 'start' creates the variable. Obviously as the code body is copied into the system heap, it must be position independent. The two routines readCode and writeCode are called whenever an access to the variable is made. For example, a *Set Mode command will call the write code entry, and *Show Sys$Mode or *Echo Mode will call the read entry.

Notice that in the body of the code variable, only XOS_ SWIs are used. This is because it is important that errors are not generated when the read or write code executes. A more rigorous version of the code above would check V after each SWI and return if it was set.

OS_AddCallBack

The next example shows the use of OS_AddCallBack; it prints ‘Run away!’ after 2 seconds:

```
DIM code 100
P%=code
[
.alarm STMFD R13!, {R14}
        SWI "XOS_WriteS"
        EQUS "Run away!"
        EQUB 10:EQUB 13:EQUB 0
        ALIGN
        LDMFD R13!, {PC}
.timer  STMFD R13!, {R0,R14}
        MOV R0, R12                     ; set up for us by BASIC bit
                                        ; R12 is not used in alarm,
                                        ; so R1 here is don’t-care
        SWI "XOS_AddCallBack"
        LDMFD R13!, {R0, PC}
]
SYS "OS_CallAfter",200,timer,alarm
```

A CallBack handler

The final example shows a CallBack handler, with a semaphore to prevent recursive CallBack; it prints ‘Run away!’ when mouse buttons are pressed.

```
DIM code 200
P%=code
[
.sema   EQUB 1
        ALIGN
.saveblock :]:P%=P%+16*4:[
.callback                               ; entered here in a privileged mode,
                                        ; with interrupts disabled
                                        ; first thing to do is enable IRQs
        TEQP PC, #3                     ; force SVC mode, IRQs on.
        SWI "XOS_WriteS"
        EQUS   "Run away!"
        EQUB 10:EQUB 13:EQUB 0
        ALIGN
        ADR R14, saveblock
        LDMIA R14, {R0-R14}^            ; most registers reloaded
        MOV R0, R0
        TEQP PC, #3+(1<<27)             ; disable IRQs for sema update
        MOV R14, #1                     ; and return
        STRB R14, sema
        LDR  R14, saveblock+15*4        ; must not allow another CallBack
                                        ; request until the stashed PC is safe
        MOVS PC, R14                    ; return, enableing IRQs etc
.events
        CMP R0, #10                     ; mouse button state change?
        MOVNES PC, R14                  ; no - run away
        STMFD R13!, {R14}
        LDRB R12, sema                  ; possibly request CallBack
        MOV R14, #0
        STRB R14, sema                  ; and disable any futher requests
        LDMFD R13!, {PC}                ; until that one serviced.
]
SYS"OS_ChangeEnvironment",7,callback,0,saveblock,0 TO ,ocall,,osave
REM Note that we aren't using R12 in the CallBack handler;
REM if this was in a module, for example, sema would be in the workspace,
REM and we would have to access it R12-relative; R12 would therefore be
REM set to be the workspace pointer on entry.
SYS "OS_ChangeEnvironment",10,events,0 TO ,oldev
*FX 14,10
REPEAT UNTILINKEY -1: REM loop until shift
*FX 13,10
SYS "OS_ChangeEnvironment",10,oldev,0
SYS"OS_ChangeEnvironment",7,ocall,,osave
REM Note that in both the above calls, the R12 values are explicitly left
REM alone, because we didn't use them earlier.
```

16 Memory Management

Introduction

This chapter describes the memory management in RISC OS. This covers memory allocation by a program or module as well as using the MEMC chip to handle how memory is mapped.

In many environments, such as BASIC and C, you can use the language's intrinsic memory allocation routines, which use the calls described in this chapter transparently. For example, refer to Wimp_SlotSize on page 3-203 of the chapter entitled *The Window Manager*.

Similarly, small, transiently loaded utilities may not require any memory over the 1024 bytes they are automatically allocated. Some programs and modules, however, will require arbitrary amounts of memory, which can be freed after use. For example, filing systems, specialised VDU drivers such as the font manager and so on. The memory manager provides simple allocation and deallocation facilities. Relocatable modules can use this manager either directly, to manipulate their own private heap, or indirectly using the module support calls.

A block of memory can be set up as a heap. This is a structure that allows arbitrary parts of the block to be allocated and freed. A program simply requests a block of a given size and is given a pointer to it by the heap manager. This block can be expanded or contracted or freed by using this pointer as a reference.

The part of the screen RAM that is not visible on the screen is also available as a temporary buffer. This memory is temporarily available because of the way that vertical scrolling is done.

One of the other memory resources available is the battery-backed CMOS RAM. This is used to hold default system parameters while the power is off. Some spare locations in CMOS RAM are reserved for users' own purposes, and for the use of expansion cards; application authors wishing to use CMOS RAM should ask Acorn for an allocation (although this will not be given if options can instead be saved to file).

The MEMC chip controls how logical addresses (those used by programs or modules) are mapped into the physical memory location to use. Numerous calls are used to control how it does this, though generally this is something that most programs would not want to do.

Overview

Heap manager

RISC OS contains a heap management system. This is used by the operating system to allocate space within the relocatable module area and also to maintain the system heap. A heap is just an area of memory from which bytes may be allocated, then deallocated for later use. An area can also be reallocated, meaning that its size changes.

The heap manager is also available to the user. You provide an area of memory which is to be used for the heap, which can be any size you require. If you are a module, then the heap would be a block within the RMA, and if you are a program, then it would be within the application space.

Thus, it would be a heap within a heap; for example a block in the RMA would be allocated by a module, and then declared as a heap. In theory, this process could continue indefinitely, but in practice this is as far as you need to go.

At the start of a heap, the heap manager sets up the heap descriptor, which is a block containing information on the limits of the heap, etc. This descriptor is updated by the heap manager when necessary.

When a block within this heap is required, a request is made to the heap manager, which returns a pointer to a suitable block of memory. The heap manager keeps a record of the total amount of memory which is free in the heap and the largest individual block which is available.

Heap fragmentation

The heap management system does not provide free-space collation. This is the technique of moving blocks of allocated memory around so as to maximise the contiguous free space and avoiding excessive fragmentation of the heap.

Also, the heap management system will never attempt to move a block within the heap, since it has no knowledge of whether the block contains pointers that need to be relocated, or whether there are any pointers to the block which need updating. Hence, unless an area of contiguous free space of the size requested is available, a request for a block will fail.

MEMC control

The MEMC chip maps logical onto physical addresses. To do this, it maintains a table of entries that map a given memory block to a particular address. Generally, the system will take care of the operation of this mapping for you. Calls are provided to allow you to read this mapping and alter it, but you should have a very good reason to do so, and be certain of what you are doing.

Screen memory

The vertical scrolling technique used under RISC OS is to change the memory location that the screen starts at. This means that part of the screen memory may be unused, depending on the screen mode and the amount of memory reserved. You can use this memory temporarily, as long as you don't cause any output that may scroll the screen. Also remember that this memory is limited to one program using it at a time, so it may not be available every time you request it. Consequently, you cannot rely on it being there when writing a module or application.

Battery-backed CMOS RAM

A block of 240 bytes of battery-backed CMOS RAM is available under RISC OS. Each location has a specific meaning and should not be directly modified unless you are sure of the meaning of the value. Many of these locations are changed indirectly using the *Configure commands. These can be found throughout this manual, in the chapter appropriate to their function.

Some bytes are not allocated, and are reserved for users and applications to use. If you want to use one or more of the application bytes, you should request a location in writing from Acorn Computers. This is so that different applications don't accidentally use the same location.

Technical Details

Guidelines on using memory efficiently

This section provides basic information on memory management by RISC OS applications. It is intended to provide some specialist knowledge to help you write efficient programs for RISC OS, and to provide some practical hints and tips.

All the information in this chapter relating to programs written in C refers to Acorn's Desktop C product.

You should follow the guidelines in this section to make the best use of available memory. The guidelines are explained in more detail on the following pages.

- **Use recovery procedures** – Your program should keep the machine operational. Don't allow your program to lock up when memory runs out; your program should indicate that it has run out of memory (with an error or warning message) and only stop subsequent actions that use more memory. Ideally, ensure that actions which free up memory have enough reserved memory to run in.
- **Return unwanted memory** – You should return any memory you have no further use for. Claiming memory then not returning it can tie up memory unnecessarily until the machine is re-booted. RISC OS has no garbage collection, so once you have asked for memory RISC OS assumes that you want it until you explicitly return it, even if your program terminates execution. Language libraries often provide you with protection from this, as long as memory is claimed from them.
- **Don't waste memory** – You should avoid wasting memory. It is a finite resource, often wasted in two ways:
  - by permanently claiming memory for infrequent operations
  - by fragmenting it, so that although there is enough unused memory, it is either in the wrong place, or it is not in large enough blocks to use.

Recovery from lack of memory

An important consideration when designing programs for RISC OS is the recovery process, not just from user errors, but also from lack of system resources.

An example of a technique that can be designed into an application is to make an algorithm more disc-based and less RAM-based on detection of lack of memory. This could allow you to continue using an application on a small machine (especially one with a hard disc) at the expense of some speed.

When implementing your code, expect the unexpected and program defensively. Be sure that when the system resources you need (memory, windows, files etc) are not available, your program can cope. Make sure that, when a document managed by your program expands and memory runs out, the document is still valid and can be saved. Don't just check that your main document expansion routines work; check that **all** routines which require memory (or in fact any system resource) fail gracefully when there is no more.

Centralising access to system resources can help: write your program as if every operating system interface is likely to return an error.

Avoiding permanent loss of memory

Permanent loss of memory is mainly a problem for applications or modules written entirely in assembly language. When interworking assembler routines with C or another high level language you should use memory handed to you by the high level language library (eg use `malloc` to get a memory area from C and pass a pointer to it as an argument to your assembler routine). The language library automatically returns such areas to RISC OS on program exit. Additional types of program requiring care to avoid memory loss are those expected to run for a long time (eg a printer spooler) and those making use of RMA directly through SWI calls.

When using the RMA for storage directly through SWI calls, especially for items in linked lists, consider using the first word as a check word containing four characters of text to identify it as belonging to your program. When a block of RMA is deallocated, the heap manager puts it back into a list of free blocks, and in so doing overwrites the first word of the block.

This technique therefore serves two purposes:

1 after your program has been run and exited, your check word can be searched for, showing up any blocks you have failed to deallocate

2 it avoids problems when accidentally referencing deallocated memory.

A typical problem of referencing deallocated blocks results from using the first word as a pointer to your program's next block, then accidentally referencing a wild pointer when it is overwritten.

You can use the following BASIC routine to search for any lost blocks:

```
100 REM > LostMemory checks for un-released blocks
110 SYS "OS_ReadDynamicArea",1 TO RMA%: RMAEnd% = RMA% + (RMA%!12)
120 FOR PossibleBlock% = RMA%+20 TO RMAEnd%-12 STEP 16
130    REM Now loop looking for "Prog"
140    IF PossibleBlock%!0 = &676F7250 THEN
150       PRINT "Block found at &";~PossibleBlock%
160    ENDIF
170 NEXT PossibleBlock%
180 END
```

When writing relocatable module initialisation code you should check that memory and other system resources are returned if initialisation is unable to complete and is going to return with V set. It is often useful to construct module finalisation code as a mirror image of initialisation code so that it can be jumped to when initialisation is going to return an error and cleaned up. A typical algorithm is:

Initialisation

```
        Claim main workspace: If error then keep this error and goto Exit3
        Claim secondary workspace: If error then keep this error and goto Exit2
        Claim tertiary workspace: If error then keep this error and goto Exit1
        Return
```

Finalisation

```
        Set kept error to null
        Release tertiary workspace
Exit1   Release secondary workspace
Exit2   Release main workspace
Exit3   Get kept error (if there was one)
        Return
```

Avoiding memory wastage

The key factor in writing programs that use memory efficiently and don't waste it is understanding the following:

- how SWI XOS_Module and SWI XOS_Heap work if you are constructing a relocatable module or are using the RMA from an application
- how C `flex` and `malloc` work when writing a C program (parts of which may be written in assembler).

This understanding will lead you to writing programs that will work in harmony with the storage allocator. See the following section for a description of C memory allocation.

The C storage manager

Understanding the C storage manager is obviously useful to writers of C. But it may also be useful to writers of assembly language for two reasons: to assist in constructing part C and part assembler programs; to assist in constructing their own memory allocation routines, both as an example algorithm and as an allocator that may be running for other applications at the same time as their own.

Normal C applications (ie those not running as modules) claim memory blocks in two main ways:

- from `malloc`
- from `flex`.

The `malloc` heap storage manager is the standard interface from which to claim small areas of memory. It is tuned to give good performance to the widest variety of programs.

In the following sections, the word *heap* refers to the section of memory currently under the control of the storage manager (usually referred to as `malloc`, or the `malloc` heap).

The flex facility is available as part of RISC_OSLib, and can be useful for claiming large areas of data space. It manages a shifting set of areas, so its operation can be slow, and address-dependent data cannot be stored in it. However, it has the following advantages:

- it doesn't waste memory by fragmenting free space
- it returns deallocated memory to RISC OS for use by other applications.

Allocation of malloc blocks

All block sizes allocated are in bytes and are rounded up to a multiple of four bytes. All blocks returned to the user are word-aligned. All blocks have an overhead of eight bytes (two words). One word is used to hold the block's length and status, the other contains a guard constant which is used to detect heap corruptions. The guard word may not be present in future releases of the ANSI C library. When the stack needs to be extended, blocks are allocated from the `malloc` heap.

When an allocation request is received by the storage manager, it is categorised into one of three sizes of blocks

- small $0 \rightarrow 64$ bytes
- medium $65 \rightarrow 512$ bytes
- large $513 \rightarrow 16777216$ bytes.

The storage manager keeps track of the free sections of the heap in two ways. The medium and large sized blocks are chained together into a linked list (overflow list) and small blocks of the same size are chained together into linked lists (bins). The overflow list is ordered by ascending block address, while the bins have the most recently freed block at the start of the list.

When a small block is requested, the bin which contains the blocks of the required size is checked, and, if the bin is not empty, the first block in the list is returned to the user. If there was not a block of the exact size available, the bin containing blocks of the next size up is checked, and so on until a block is found. If a block is not found in the bins, the last block (highest address) on the overflow list is taken. If the block is large enough to be split into two blocks, and the remainder is a usable size (> 12 including the overhead) then the block is split, the top section returned to the user and the remainder, depending on its size, is either put in the relevant bin at the front of the list or left in the overflow list.

When a medium block is requested, the search ignores the bins and starts with the overflow list. This is searched in reverse order for a block of usable size, in the same way as for small blocks.

When a large block is requested, the overflow list is searched in increasing address order, and the first block in the list which is large enough is taken. If the block is large enough to be split into two blocks, and the size of the remainder is larger than a small block (> 64) then the block is split, the top section is returned to the overflow list, and bottom section given to the user.

Should there not be a block of the right size available, the C storage manager has two options:

1 Take all the free blocks on the heap and join adjacent free blocks together (coalescing) in the hope that a block of the right size will be created which can then be used

2 Ask the operating system for more heap, put the block returned in the overflow list, and try again.

The heap will only be coalesced if there is at least enough free memory in it to make it worthwhile (ie four times the size of the requested block, and at least one sixth of the total heap size) or if the request for more heap was denied. Coalescing causes the following:

- the bins and overflow list are emptied;
- the heap is scanned;
- adjacent free blocks are merged;
- the free blocks are scattered into the bins and overflow list in increasing address order.

Deallocation of malloc blocks

When a block is freed, if it will fit in a bin then it is put at the start of the relevant bin list, otherwise it is just marked as being free and effectively taken out of the heap until the next coalesce phase, when it will be put in the overflow list. This is done because the overflow list is in ascending block address order, and it would have to be scanned to be able to insert the freed block at the correct position. Fragmentation is also reduced if the block is not reusable until after the next coalesce phase. It is worth noting that deallocating a block and then reallocating a block of the same size can not be relied upon to deliver the original block.

Reallocation of malloc blocks

You should be cautious when using `realloc`. Reallocating a block to a larger size will usually require another block of memory to be used and the data to be copied into it. This means that you cannot use the whole of the heap as both blocks need to be present at the same time.

If consecutive calls keep increasing the block size until all memory is used up, then only about a third of the heap is likely to be available in one block. A typical course of events is:

1 The first block is present (block A).

2 It is extended to a larger sized block (block B). Block A must still be present (see above).

3 It is again extended to a larger sized block (block C). Block B must still be present (see above). However, block A also still exists because it is too small to use, and cannot be coalesced with another block because block B is in the way.

Wimp slots and the C flex system

A typical C application running under the Wimp has a single contiguous application area (wimp slot) into which are placed the following:

- program image
- stack
- static data
- `malloc` data.

The initial wimp slot size is set by the size of the Next slot (in the Task display window) when the application is started, or by *WimpSlot commands in the !Run file associated with the C application. If the `malloc` heap is full and the operating system has free memory, the wimp slot grows, raising its highest address. Once enlarged by `malloc`, the wimp slot never reduces again until program termination.

The application area is used as follows:

low memory:	the application image
	the static data
high memory:	the `malloc` heap

The stack is allocated on the heap, in 4K (or as big as needed) chunks: the ARM procedure call standard means that disjoint extension of the stack is possible. The only other use that the ANSI C library makes of the `malloc` heap is in allocating file buffers, but even this usage can be prevented by making the appropriate calls to the ANSI C

library buffer handling facilities (`setvbuf`). The operation of the `malloc` heap is described above and is designed to provide good performance under heavy use. Its design is such that small blocks can be allocated and freed rapidly.

Any `malloc` heap tends to fragment over time. This is particularly serious in the following circumstances:

- no virtual memory
- multitasking – if memory is not in use, it should be handed to other applications
- if a program runs out of memory it must not crash, but must recover and continue.

These are just the conditions under which a desktop application operates!

Because of this, the flex facilities are available as part of RISC_OSLib (the RISC OS-specific C library provided with Desktop C). These provide a shifting heap, intended for the allocation of large blocks of memory which might otherwise destroy the structure of a `malloc`-style heap.

Flex works by increasing the size of the application area, using space above that reserved for use by `malloc`. When the `malloc` heap grows, flex areas are shifted. The benefits of using flex can be seen in Draw, Paint and Edit, which are all written in C using early versions of RISC_OSLib. Their application areas expand when new files are added, contract when files are discarded, and do not suffer from needless incremental application area growth over time.

The implementation of flex is quite simple. There is no free list as memory is shifted whenever a block is destroyed or changed in size. New blocks are always allocated at the top. When blocks are deallocated or resized, those above are moved. This means that deallocating or changing the size of a block can take quite a long time (proportional to the sum of the sizes of the blocks above it in memory). Flex is also not recommended for allocation of small blocks. Its other limitation is that as flex blocks can be shifted, you should not use them for address-dependent data (eg pointers or indirected icon data).

In addition to the facilities described above, RISC_OSLib also provides an obsolete `malloc`-like allocator of non-shifting blocks called heap.

Two facilities are provided, because no one storage manager can solve all problems in the absence of Virtual Memory. A program which works adequately with `malloc` should feel no compulsion to use anything else. The use of flex, however, particularly in desktop applications such as editors (which are likely to be resident on the desktop for a long period of time) can go a long way towards improving their memory usage.

Using memory from relocatable modules

Relocatable modules should use memory from three sources: the supervisor stack; the RMA; and application workspace. Use of pc-relative written data should be avoided as it makes a module unsuitable to ROM, unsuitable for multiple instantiation, and permanently reserves space, possibly only for occasional use.

The supervisor stack is small and not extendable, so care must be taken to use this resource very economically.

The RMA is the standard source of workspace for any of the non-user mode routines contained in a module. Care must be taken to deallocate unwanted blocks – the marker word hint described earlier in this chapter may be useful. C malloc uses RMA when called from non-user mode.

Application workspace only belongs to a module when referenced from module user mode code running as the sole current application (with RISC OS desktop multitasking halted) or when running as a RISC OS application having dealt with the `Service_Memory (&11)` service call (sent round by the wimp when your program issues `SWI Wimp_Initialise`) to keep application workspace.

Never access your application's workspace from an interrupt routine. During interrupts, the state of the application area is effectively random. Since your interrupt routine could execute at any time, it could happen while some other application is switched in. If this did happen, and the interrupt routine updated application space, then some other application could be affected. To get around this problem, allocate some RMA space for your interrupt routine to use when it needs to; this memory will be visible when your application is running. Remember to free up the RMA space when you've finished with it.

Using memory from relocatable modules written in C

There are additional points you should note if you are writing modules in C (although most of the points made above apply equally well – particularly the preceding paragraph).

All memory allocated by `malloc` comes from the RMA when your program is executing in non-user mode. So remember to free it up when you've finished with it. If your module allocates any RMA blocks by calling SWI XOS_Module directly, the C run-time system does not clear them out when your module finalises, so make sure you do!

There are two sets of `atexit()` routines, the ones which you registered during initialisation ie before your module was entered via the `main()` entry point (because the module was `RMRun` for instance), and the ones you registered after. The ones

registered before will be executed when your module is finalised – this is how to clear up after yourself; the ones after will be called when your module exits from being run, ie when `main()` terminates.

When you are writing a C module, use `exit()`, not SWI XOS_Exit.

When executing as C module SVC mode code (during initialisation, finalisation, service or interrupt entry) your stack will be small. Also, your stack, unlike when in USR mode (ie running as an application) will not extend dynamically. It is therefore very important to be extremely economical with stack space; eg avoiding large auto arrays, using `malloc` where larger spaces are required, and freeing claimed memory at the routine end.

Static variables (and arrays etc.) in a C module are extant for the lifetime of the module, ie the entire time it is loaded. If they are only needed when it is running as an application, then they should be claimed using `malloc` instead.

Heap Manager

The heap is controlled by a single SWI, OS_Heap (page 1-377 onwards). This has a reason code and can perform the following operations:

Reason code	Meaning
0	Initialise heap
1	Describe heap
2	Allocate a block from a heap
3	Free a block
4	Change the size of a block
5	Change the size of a heap
6	Read the size of a block

Internal format of the heap

A description of the structure used by the heap manager is given below. It should be noted that this structure is not guaranteed to be preserved between releases of the software and should not be relied upon. It is given purely for advanced programmers who may want to interpret the current state of the heap when testing and debugging their own code.

The heap descriptor is a block of four words:

&00	Special heap word
&04	Free list offset
&08	Heap base offset
&0C	Heap end offset

Figure 16.1 Format of heap descriptor

The 'special' heap word contains a pattern which distinguishes correct heap descriptors. The pattern is made up of the characters 'Heap' – which is &70616548 in hex.

All other words are offsets into the heap. This means that the heap is relocatable unless you place non-relocatable information in it.

The free list offset is an offset to the first free block in the heap, or zero if there are no free blocks. If the word is non-zero, the first free block is at address:

heap start + free list offset + 4

The other entries are offsets from the start of the heap which refer to boundaries within the heap structure. The heap is delimited as follows:

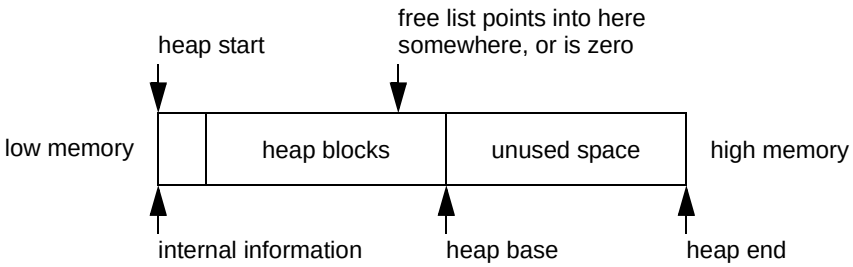


Figure 16.2 How the heap is delimited

Blocks in the free list have information in the first two words as follows:

- Word 0 is the link to the next free block or 0 if at the end
- Word 1 is the size of this block (including these two words)

Allocated blocks start with a word which holds the size of the allocated block. The pointer returned by SWI OS_Heap when a block is allocated actually points to the second word which is the start of the memory available.

Allocation forces the block size to be a multiple of eight, to ensure that no matter what you do, the fragments can always be freed. Therefore, the minimum size of area that can be initialised is 24 bytes (16 for the fixed information and 8 for a block).

Logical memory map

The organisation of the logical address space is **currently** as follows:

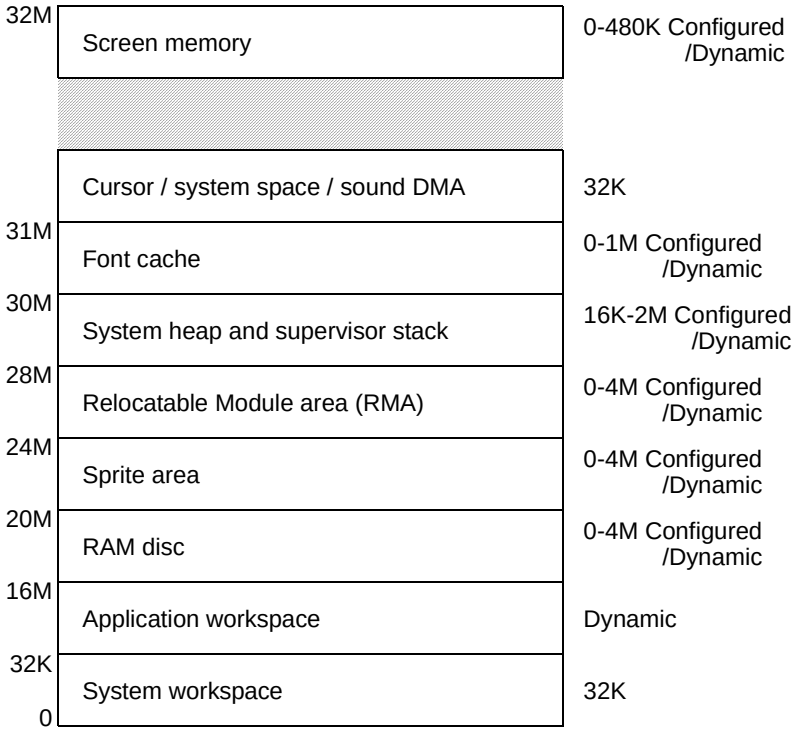


Figure 16.3 Typical logical memory map

You must not assume that any of the above addresses will remain fixed (save for the base of application workspace). There are defined calls to read any addresses you need, and you must use them.

Setting up the memory map

The memory map is set up on hard reset as follows:

- The permanent 32K allocations for system workspace at addresses &0000000 and &1F00000 (31Mbytes) are made, as well as some other fixed allocations (such as an initial part of the system heap).

- Then space is allocated to the various adjustable size regions, such as the screen, the system heap, the RMA, etc. Some of these have an absolute configured size, such as the screen. This is allocated in full. For other regions (such as the system heap and RMA), the configured size is the amount of free space that will be left; these only have a minimal allocation made at this stage.
- The rest of memory is then allocated to the application workspace, from address &8000 up.
- System ROM and expansion card modules are then initialised.
- Finally, the regions that have a configured free space get allocated. First they are shrunk as far as possible (to ensure as close to 0 bytes free as possible), then a block of the configured size is requested and freed, so that the heaps contain as close to the configured free space as possible.

Example memory allocation

Here is an example of how memory might be allocated given some typical RAM size allocations on an A310 (8K page size):

Area	Pages	Page size	Total
FontSize	20	4K	80K
RamFsSize	0	8K	0
RMASize	16	8K	128K
ScreenSize	20	8K	160K
SpriteSize	10	8K	80K
SystemSize	4	8K	32K+32K
System workspace			32K
Cursor etc. workspace			32K
Total			576K
Application area	1024K – 576K = 448K		

Note that although the FontSize is configured in units of 4K, it is always allocated in multiples of the MEMC page size. A configured screen size of 0 means ‘default for this machine’, which is 160K on an A310 (see *Configure ScreenSize).

As outlined above, the size of the system area (at 28M) is shrunk as far as possible after all module initialisation and then ‘n’ extra pages are added. 8K of this is used for the system stack. The rest is for OS variable storage (eg alias variables) and module information. The configured amount is added to the 32K initially allocated.

Altering the memory map

While no application is running (ie in the supervisor prompt), the memory map can be altered as required. For example, if you load a module from disc and the RMA isn’t big enough to hold it, the size of the RMA will be increased by an appropriate amount. The

OS can only do this when there is no application active, as the extra memory has to be taken from the application workspace. Most programs don't react too kindly to large areas of their memory allocation disappearing.

Under an environment such as the desktop, multiple applications are run concurrently. The currently running application is mapped into memory at &8000. Desktop applications periodically return control to the Window Manager (or *Wimp*) by calling the SWI Wimp_Poll; at this point the Wimp may decide to swap to another application. In doing so, it maps out the current application, and maps the new application into that space. Thus every application is given the illusion that it is the only one in the system.

Page size

The SWI OS_ReadMemMapInfo (page 1-391) returns the page size used in the system and the number of pages present. For more details of page sizes, see the section entitled *Page size* on page 1-19.

Controlling memory allocation

OS_ChangeDynamicArea (page 1-384) allows control of the space allocated to the system heap, RMA, screen, sprite area, font cache and RAM filing system. Any space left over is the application space by default. Any of these settings can be read with OS_ReadDynamicArea (page 1-396). OS_ReadRAMFsLimits (page 1-390) will read the range of bytes used by the RAM filing system. The size of it can be set in CMOS RAM using *Configure RamFsSize. See also *Configure RMASize and *Configure SystemSize.

Memory protection

You have read/write access to much of the logically mapped RAM. There are exceptions, such as the 32K system workspace at &1F00000 (31M), the RAM disc, and the font cache. More areas may become protected in future releases of RISC OS. The **only** areas normal applications should directly access are the application workspace and the RMA. Specialist programs may access other areas of memory; for example a set of extension graphics primitives may write directly to the screen (of course reading the screen's base address using a defined call: in this case OS_ReadVduVariables). In general, though, it is **very dangerous** to write to these other areas, or rely on certain locations containing given information, as these are subject to change. You should always use OS routines to access operating system workspace.

OS_ValidateAddress (page 1-386) will check a range of logical addresses to see if they are mapped into physical memory.

Changing the logical map

The mapping that MEMC maintains from logical to physical address space can be read with OS_ReadMemMapEntries (page 1-392). This gives a list of physical addresses for a matching set of logical page numbers.

The reverse operation, OS_SetMemMapEntries (page 1-394) will write the mapping inside MEMC. Note that this is an extremely dangerous operation if you are not sure what you are doing.

OS_UpdateMEMC (page 1-373) is a lower level operation that alters the bits in the MEMC control register.

Screen memory

The screen workspace is at the end of logical memory, adjacent to the physical RAM which is mapped onto those addresses. This means that there are two adjacent copies of the screen memory.

Writing to the screen

The display is normally set up by RISC OS's VDU drivers, which write to the logical memory.

You can read various VDU and mode variables to find the addresses used for this. In particular, the ScreenStart VDU variable give the logical address of the base of screen memory, the ScreenSize mode variable gives the amount of memory used by the current mode (and hence the logical address of the top of screen memory), and the TotalScreenSize VDU variable gives the amount of memory allocated to the display.

The screen-size is configurable in units of one page. Hence for a 20K screen on a 400 series machine, 32K will have to be used since it is the next highest multiple of 32K. For an 80K screen, 96K would be used, etc. In addition, if you want to use multiple banks of screen memory (eg for animation), enough memory must be reserved for each bank.

Because the total screen memory is often much more than is required at a given time, the SWI OS_ClaimScreenMemory (page 1-388) is provided so you can claim the 'extra' RAM for short periods. It can be used as a buffer, in a data transfer operation, for example.

Displaying the screen

The display is output by MEMC using DMA to access the area of physical memory corresponding to the logical area used by the VDU drivers, and passing this area's contents to VIDC for conversion to a video signal. The area is treated as a circular buffer.

Video DMA is controlled by the physical addresses in various MEMC registers. At the start of a frame the Vptr register (ie the video DMA pointer) is set to the address in the Vinit register – which normally corresponds to the logical address in the ScreenStart VDU variable. Each read Vptr is incremented, unless it has reached the end of the buffer (as delimited by the Vend register), in which case Vptr is reset to the start of the buffer (given by the Vstart register).

Summary and hardware scrolling

This section gives two diagrams to illustrate the above; they also show how hardware scrolling is implemented.

For an unscrolled screen, access to screen memory takes place as follows:

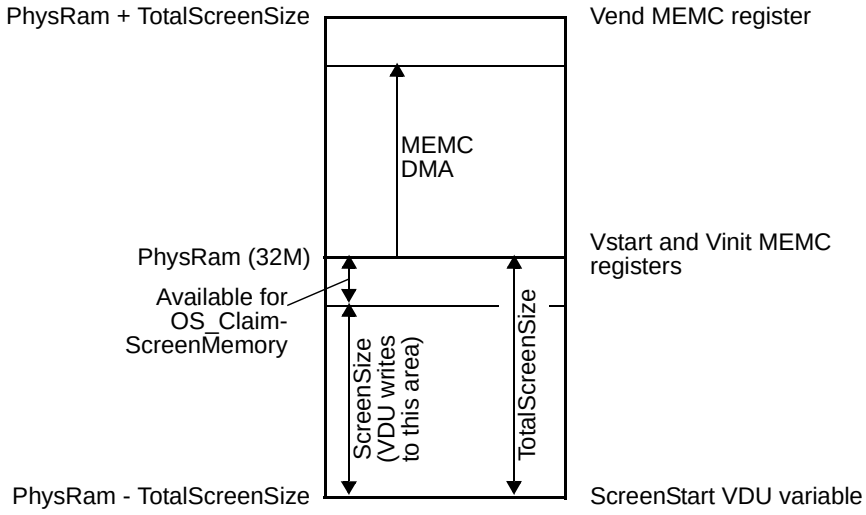


Figure 16.4 Screen memory

Vertical hardware scrolling is implemented by altering MEMC's Vinit register. At the same time the ScreenStart VDU variable must be altered so that the VDU drivers write to the corresponding location in logical memory. This means that with larger amounts of scrolling, a part of the logical area to which the VDU drivers are writing is in fact an area of physical memory:

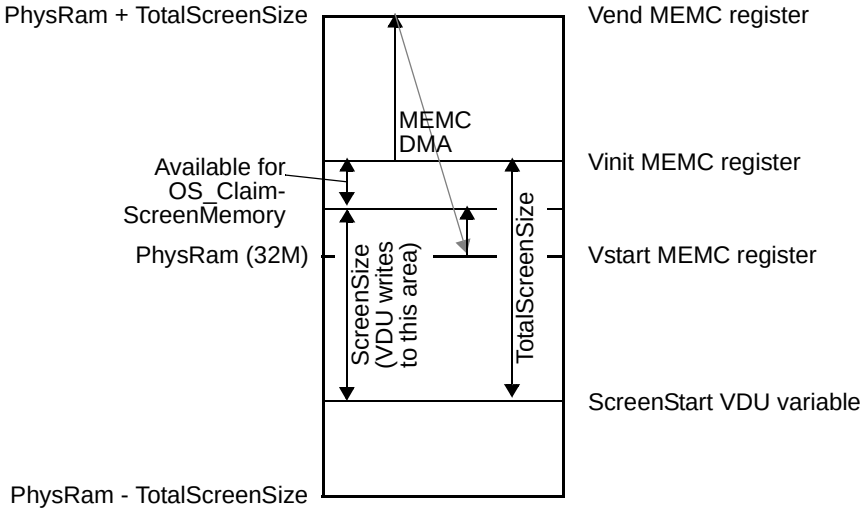


Figure 16.5 Screen memory after hardware scrolling

Normally hardware scrolling is performed automatically, and you don't have to concern yourself with it. However, if you need to implement it yourself – for a game, for instance – you should be in a privileged processor mode, so you can both alter MEMC's Vinit register and write to physical memory.

Non-volatile memory (CMOS RAM)

240 bytes of non-volatile memory are provided. The majority of these bytes are reserved for, or used by Acorn. Some bytes are reserved for each expansion card; before using these, see the section entitled *CMOS RAM* on page 4-133. There are also bytes reserved for the user; you must not use these in any distributed product. Finally, there are bytes reserved for applications; for an allocation, contact Acorn in writing, but see first the section entitled *CMOS RAM bytes* on page 4-553.

CMOS usage is subject to change in different versions of RISC OS, and your application should not assume the location of any particular information.

OS_Byte 161 (page 1-369) allows you to read the CMOS memory directly, while OS_Byte 162 (page 1-371) can write to it.

RISC OS 3

The full usage of CMOS RAM in RISC OS 3 is given below. Locations marked '†' are unused in RISC OS 2, and are therefore reserved for Acorn use. Locations marked '‡' have differing usage in RISC OS 2, and you should see page 1-366 for details:

Location	Function	
0	Econet station number (not directly configurable)	
1	Econet file server station id (0 ⇒ name configured)	
2	Econet file server net number (or first char of name – rest in bytes 153 - 157)	
3	Econet printer server station id (0 ⇒ name configured)	
4	Econet printer server net number (or first char of name – rest in bytes 158 - 172)	
5	Default filing system number	
6 - 7 †	*Unplug for ROM modules: 16 bits for up to 16 modules	
8 - 9	Reserved for Acorn use	
10	Screen info:	
	Bits 0 - 3 ‡	reserved for Acorn use
	Bit 4	TV interlace (first *TV parameter)
	Bits 5 - 7	TV vertical adjust (signed three-bit number)
11	Shift, Caps mode:	
	Bits 0 - 2	reserved for Acorn use
	Bits 3 - 5	001 ⇒ ShCaps, 010 ⇒ NoCaps, 100 ⇒ Caps
	Bit 6 - 7	reserved for Acorn use
12	Keyboard auto-repeat delay	
13	Keyboard auto-repeat rate	
14	Printer ignore character	
15	Printer information:	
	Bit 0	reserved for Acorn use
	Bit 1	0 ⇒ Ignore, 1 ⇒ NoIgnore
	Bits 2 - 4	serial baud rate (0=75,…,7=19200)
	Bits 5 - 7	printer type
16	Miscellaneous flags:	
	Bit 0	reserved for Acorn use
	Bit 1	0 ⇒ Quiet, 1 ⇒ Loud
	Bit 2	reserved for Acorn use
	Bit 3	0 ⇒ Scroll, 1 ⇒ NoScroll
	Bit 4	0 ⇒ NoBoot, 1 ⇒ Boot
	Bits 5 - 7	serial data format (0…7)
17	NetFiler:	
	Bit 0	FS list sorting mode: 0 ⇒ by name, 1 ⇒ by number
	Bit 1	library type: 0 ⇒ default library returned by

		file server, 1 ⇒ $.ArthurLib
	Bits 2 - 3	FS list display mode: 0 ⇒ large icons, 1 ⇒ small icons, 2 ⇒ full info, 3 reserved
	Bits 4 - 7	reserved for Acorn use
18 - 19 †	*Unplug for ROM modules: 16 bits for up to 16 modules	
20 - 21 †	*Unplug for extension ROM modules: 16 bits for up to 16 modules	
22 †	Wimp double-click move limit	
23 †	Wimp auto-menu delay	
24 †	Territory	
25 †	Printer buffer size	
26 †	IDE disc auto-spindown delay	
27 †	Wimp menu drag delay	
28 †	FileSwitch options:	
	Bit 0	truncate names: 0 ⇒ give error, 1 ⇒ truncate
	Bit 1	DragASprite: 0 ⇒ don't use, 1 ⇒ use
	Bit 2	interactive file copy: 0 ⇒ use, 1 ⇒ don't use
	Bit 3	Wimp's use of dither patterns on desktop: 0 ⇒ don't use, 1 ⇒ use
	Bit 4	Shift toggle size behaviour: 0 ⇒ standard RISC OS 3 behaviour, 1 ⇒ never obscure icon bar
	Bit 5	reserved for Acorn use
	Bits 6 - 7	state of last shutdown: 0 ⇒ don't care, 1 ⇒ failed, 2 ⇒ due to power loss, 3 ⇒ undefined
29	Reserved for Acorn use	
30 - 45	**Reserved for the user**	
46 - 79	**Reserved for applications**	
80 - 111	Reserved for RISC iX	
112 - 127	Reserved for expansion card use	
128 - 129	Current year	
130 - 131	Reserved for Acorn use	
132	DumpFormat and Tube expansion card:	
	Bits 0,1	control character print control: 0 ⇒ print in GSTrans format, 1 ⇒ print as a dot, 2 ⇒ print decimal inside angle brackets, 3 ⇒ print hex inside angle brackets
	Bit 2	treat top-bit-set characters as valid if set
	Bit 3	AND character with &7F in *Dump
	Bit 4	treat TAB as print 8 spaces
	Bit 5	Tube expansion card enable

	Bits 6,7	Tube expansion card slot (0 - 3)
133 ‡	Sync, monitor type, some mode information:	
	Bits 0, 7	0 ⇒ vertical sync, 1 ⇒ composite sync, 3 ⇒ auto sync)
	Bit 1	reserved for Acorn use
	Bits 2 - 6	monitor type: 0 ⇒ 0, 1 ⇒ 1, …, 31 ⇒ auto
134	FontSize in units of 4K	
135 - 137	ADFS use	
138	Allocated to CDROMFS	
139 †	TimeZone in 15min offsets from UTC, stored as signed 2's complement number (RISC OS 3 version 3.10 onwards)	
140 ‡	Desktop features:	
	Bit 0	3D: 0 ⇒ standard RISC OS 2 look, 1 ⇒ 3D
	Bits 1 - 7	reserved for Acorn use
141 †	Currently selected printer, stored as printer number within current PrData file	
142	Allocated to Twin	
143	Screen size, in pages	
144	RAM disc size, in pages	
145	System heap size to add after initialisation, in pages	
146	RMA size to add after initialisation, in pages	
147	Sprite size, in pages	
148	SoundDefault parameters:	
	Bits 0 - 3	channel 0 default voice
	Bits 4 - 6	loudness (0 - 7 ⇒ &01, &13, &25, &37, &49, &5B, &6D, &7F)
	Bit 7	loudspeaker enable
149 - 152	Allocated to BASIC Editor	
153 - 157	Printer server name	
158 - 172	File server name	
173 - 176	*Unplug for ROM modules: 32 bits for up to 32 modules	
177 - 180	*Unplug for expansion card modules: 4 × 8 bits for up to 8 modules per card	
181 - 184	Wild card for BASIC editor	
185	Configured language	
186	Configured country	
187	VFS	
188	Miscellaneous:	
	Bits 0 - 1	ROMFS Opt 4 state
	Bit 2 †	cache icon enable state
	Bits 3 - 5 †	screen blanker time: 0 ⇒ off, 1 ⇒ 30s, 2 ⇒ 1min, 3 ⇒ 2mins, 4 ⇒ 5mins, 5 ⇒ 10mins, 6 ⇒ 15mins, 7 ⇒ 30mins

		Bit 6 †	screen blanker/Wrch interaction: 0 ⇒ ignore Wrch, 1 ⇒ Wrch unblanks screen
		Bit 7 †	hardware test disable: 0 ⇒ full tests, 1 ⇒ disable long tests at power-up
189 - 192	Winchester size		
193	Protection state:		
		Bit 0	Peek
		Bit 1	Poke
		Bit 2	JSR
		Bit 3	User RPC
		Bit 4	OS RPC
		Bit 5	Halt
		Bit 6	GetRegs
194	Mouse multiplier		
195 †	Miscellaneous:		
		Bit 0	AUN BootNet: 0 ⇒ disabled, 1 ⇒ enabled
		Bit 1	reserved for Acorn use
		Bit 2	type of last reset: 0 ⇒ ordinary, 1 ⇒ CMOS reset (RISC OS 3 version 3.10 onwards)
		Bit 3	power saving: 0 ⇒ disabled, 1 ⇒ enabled
		Bit 4	mode and wimp mode: 0 ⇒ use byte 196, 1 ⇒ auto
		Bit 5	cache enable for ARM3
		Bit 6	broadcast protocols enable
		Bit 7	colour hourglass enable
196 ‡	Mode and Wimp mode		
197	Wimp flags		
198	Desktop state:		
		Bits 0, 1	Filer display mode: 0 ⇒ large icons, 1 ⇒ small icons, 2 ⇒ full info, 3 reserved
		Bits 2, 3	Filer sorting mode: 0 ⇒ sort by name, 1 ⇒ sort by type, 2 ⇒ sort by size, 3 ⇒ sort by date
		Bit 4 †	force option (1 ⇒ force)
		Bit 5	confirm option (1 ⇒ confirm)
		Bit 6	verbose option (1 ⇒ verbose)
		Bit 7 †	newer option (1 ⇒ newer)

199	ADFS directory cache size	
200 - 207	FontMax, FontMax1 - FontMax7	
208 †	SCSIFS flags	
	Bits 0 - 2	number of discs (0 - 4)
	Bits 3 - 5	default drive – 4
	Bits 6,7	reserved
209 †	SCSIFS file cache buffers (must be 0)	
210 †	SCSIFS directory cache size	
211 - 214 †	SCSIFS disc sizes (their maps' sizes / 256)	
224 - 238	Reserved for RISC iX	
239 †	CMOS RAM checksum	

The checksum must be correct for some of the above locations to have effect. See the documentation of OS_Byte 162 on page 1-371 for more details.

RISC OS 2

Locations marked '†' above are unused in RISC OS 2, and are therefore reserved for Acorn use. Locations marked '‡' above have this differing usage in RISC OS 2:

Location	**Function**	
10	Screen info:	
	Bits 0 - 3	Configured screen mode, held in 5 bits, with the fifth bit in bit 1 of byte 133
133	Sync, monitor type, some mode information	
	Bit 0	0 ⇒ vertical sync, 1 ⇒ composite sync
	Bit 1	top bit of configured mode (rest held in byte 10)
	Bits 2 - 3	monitor type
	Bits 4 - 7	reserved for Acorn use
140	PrinterDP state:	
	Bit 0	print line feeds: 0 ⇒ do, 1 ⇒ don't
	Bits 1 - 2	strip control: 0 ⇒ monochrome, 1 ⇒ grey scale, 2 ⇒ colour, 3 reserved
	Bit 3	feed: 0 ⇒ auto feed, 1 ⇒ manual feed
	Bit 4	print quality: 0 ⇒ draft, 1 ⇒ NLQ
	Bits 5 - 6	halftone type: 0 ⇒ small, 1 ⇒ large, 2 ⇒ dithered, 3 reserved
	Bit 7	reserved for Acorn use
196	Wimp mode (actual mode EOR &0C)	

Service Calls

Service_MemoryMoved (Service Call &4E)

OS_ChangeDynamicArea has just finished

On entry

R1 = &4E (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is made whenever OS_ChangeDynamicArea (page 1-384) has just finished. It is used by the Wimp to tidy up and should never be claimed.

Service_ValidateAddress (Service Call &6D)

OS_ValidateAddress has been called with an unrecognised area

On entry

R1 = &6D (reason code)
R2 = start of area (value passed in R0 on entry to OS_ValidateAddress)
R3 = end of area + 1 (value passed in R1 on entry to OS_ValidateAddress)

On exit

R1 = 0 to indicate area is valid; else preserved to pass on

Use

This call is intended for internal use only. Application modules should not need to claim or issue this service.

SWI Calls

OS_Byte 161 (SWI &06)

Read battery-backed CMOS RAM

On entry

R0 = 161
R1 = RAM location

On exit

R0, R1 preserved
R2 = contents of location

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call provides read access to any of the locations in the battery-backed CMOS RAM. For example, this call may be used by a module to read a default configuration parameter. Moreover, this parameter could be examined by the user using the *Status command, if the module provides a suitable entry in its command decoding table. See the section entitled *Help and command keyword table* on page 1-216 for more details.

Related SWIs

OS_Byte 162 (page 1-371)

Related vectors

ByteV

OS_Byte 162 (SWI &06)

Write battery-backed CMOS RAM

On entry

R0 = 162
R1 = RAM location
R2 = value to be written

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 writes to any of the locations in the battery backed RAM with the exception of location zero, which is protected. In doing so the CMOS RAM checksum is maintained but not recalculated; ie it will remain in the same state of correctness as before this call. To keep the checksum correct, you should always use this call to write to the CMOS RAM, and never write to the checksum location. This call can take a comparatively long time to return (eg 20ms).

Related SWIs

OS_Byte 161 (page 1-369)

Related vectors

ByteV

OS_UpdateMEMC (SWI &1A)

Read or alter the contents of the MEMC control register

On entry

R0 = new bits in field
R1 = field mask

On exit

R0 = previous bits in field
R1 = previous field mask

Interrupts

Interrupts are disabled
Fast interrupts are disabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered because interrupts are disabled

Use

The memory controller (MEMC) chip is a write-only device. The operating system maintains a software copy of the current state of the control register and OS_UpdateMEMC updates MEMC from the software state. To allow the programming of individual bits the call takes a field and a mask. The new MEMC value is:

```
newMemC     = (oldMEMC AND NOT R1) OR (R0 AND R1)
R0          = oldMEMC
```

So to read the contents without altering them, R0 and R1 should both be zero. To set them to *n*, R0 = *n* and R1=&FFFFFFFF.

Related SWIs

None

Related vectors

None

OS_Heap (SWI &1D)

Perform various operations on the heap

On entry

R0 = reason code
R1 = pointer to heap
R2 = pointer to block (if relevant to reason code)
R3 is reason code dependent

On exit

R0, R1 preserved
R2 and R3 are reason code dependent

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 performs various operations on the heap, depending on the reason code passed in R0:

R0	Meaning	Page
0	Initialise heap	1-377
1	Describe heap	1-378
2	Get heap block	1-379
3	Free heap block	1-380
4	Extend or shrink heap block	1-381

R0	Meaning	Page
5	Extend or shrink heap	1-382
6	Read block size	1-383

Related SWIs

None

Related vectors

None

OS_Heap 0 (SWI &1D)

Initialise heap

On entry

R0 = 0 (reason code)
R1 = pointer to heap to initialise
R3 = size of heap

On exit

R0, R1, R3 preserved

Use

This call checks the given heap pointer, and then writes a valid descriptor into the heap it points at. The heap is then ready for use. The value given for R1 must be word-aligned and less than 32Mbytes (ie must point to an area of logical RAM). R3 must be a multiple of four and less than 16Mbytes.

OS_Heap 1 (SWI &1D)

Describe heap

On entry

R0 = 1 (reason code)
R1 = pointer to heap

On exit

R0, R1 preserved
R2 = largest available block size
R3 = total free

Use

This call returns information on the space available in the heap. An error is returned if the heap is invalid. This may be for any of the following reasons:

- the heap descriptor is corrupt
- the information within the heap is not sensible
- R1 does not point to a heap

OS_Heap 2 (SWI &1D)

Get heap block

On entry

R0 = 2 (reason code)
R1 = pointer to heap
R3 = size required in bytes

On exit

R0, R1 preserved
R2 = pointer to claimed block or zero if allocation failed
R3 preserved

Use

This allocates a block from the heap. An error is returned if the allocation failed for any of the following reasons:

- there is not a large enough block left in the heap
- the heap has been corrupted
- R1 does not point to a heap

OS_Heap 3 (SWI &1D)

Free heap block

On entry

R0 = 3 (reason code)
R1 = pointer to heap
R2 = pointer to block

On exit

R0 - R2 preserved

Use

This checks that the pointer given refers to an allocated block in the heap, and deallocates it. Deallocation tries to join free blocks together if at all possible, but if the block being freed is not adjacent to any other free block it is just added to the list of free blocks. An error is returned if the deallocation failed which may be because:

- R1 does not point to a heap
- the heap descriptor or heap was corrupted
- R2 does not point to an allocated block in the heap.

OS_Heap 4 (SWI &1D)

Extend or shrink heap block

On entry

R0 = 4 (reason code)
R1 = pointer to heap
R2 = pointer to block
R3 = required size change in bytes (signed integer)

On exit

R0, R1 preserved
R2 = new block pointer, or –1 if heap block extended to size 0 (or less)
R3 preserved

Use

This attempts to enlarge or shrink the given block in its current position if possible, or, if this is not possible, by reallocating and copying it. Note that if the block has to be moved, it is your responsibility to note this (by the fact that R2 has been altered), and to perform any necessary relocation of data within the block.

OS_Heap 5 (SWI &1D)

Extend or shrink heap

On entry

R0 = 5 (reason code)
R1 = pointer to heap
R3 = required size change in bytes (signed integer)

On exit

R0, R1 preserved
R3 preserved, or amount of bytes heap shrunk by if requested shrink failed

Use

This updates the heap size information to take account of the new size. If the heap cannot shrink as far as requested – because of data that has already been allocated – it will shrink as far as possible, set R3 to the amount by which it shrank, and return an error.

OS_Heap 6 (SWI &1D)

Read block size

On entry

R0 = 6 (reason code)
R1 = pointer to heap
R2 = pointer to block

On exit

R0 - R2 preserved
R3 = current block size

Use

This reads the size of a block in the specified heap. This includes any overheads associated with the block, and so will be larger, for example, than the required size of a block newly created with OS_Heap 2. An error is returned if the heap or the block could not be found.

OS_ChangeDynamicArea (SWI &2A)

Alter the space allocation of a dynamic area

On entry

R0 = area to alter
R1 = amount to move in bytes (signed integer)

On exit

R0 = preserved
R1 = number of bytes moved (unsigned integer)

Interrupts

Interrupts are disabled in critical periods, but otherwise in the caller's state
(Under RISC OS 2 interrupts are disabled throughout the call)
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ChangeDynamicArea allows the space allocated to an area to be altered in size by removing or adding workspace from the application workspace.

The area to be altered depends on R0 as follows:

Value of R0	Area to alter
0	system heap
1	RMA
2	screen area
3	sprite area
4	font cache
5	RAM filing system

The amount to move is given by the sign and magnitude of R1:

+ve means **enlarge** the selected area by at least the given amount
–ve means **shrink** the selected area by no more than the given amount

If the amount to be moved is not an exact number of pages, it is rounded up (ie in the +ve direction) to the next number of pages.

Note that normally, this cannot be used while the application work area is being used; for example when BASIC is active outside the RISC OS desktop. An attempt to do so will result in a 'Memory in use' error. (In fact, when this call is made, the OS passes a service call round to modules, which can veto the change if they can't handle it correctly. See *Service_Memory (Service Call &11)* on page 3-66 and *Service_MemoryMoved (Service Call &4E)* on page 1-367 for more details.

Any area size change will fail if the new size is smaller than the current requirements, but will shrink the area as far as it can. If you need to release as much space as possible from an area, try to reduce its size by 16 Mbytes.

Expanding, on the other hand, does nothing if it can't move enough. In this case, if you asked for the extra space you probably need it all; RISC OS assumes that half the job is no use to you.

This SWI also does an UpCall, to enable programs running in application workspace to allow movement of memory. If the UpCall is claimed when the application is running in application workspace, the memory movement is allowed to proceed. For full details see *OS_UpCall 257 (SWI &33)* on page 1-198.

An error is returned if not all the bytes were moved, or if application workspace is being used – ie an application is active.

Related SWIs

OS_UpCall 257 (page 1-198), OS_ReadDynamicArea (page 1-396)

Related vectors

None

OS_ValidateAddress (SWI &3A)

Check that a range of addresses are in logical RAM

On entry

R0 = minimum address
R1 = maximum address

On exit

R0, R1 preserved
C flag is clear if the range is OK, set otherwise

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 SWI checks the address range between R0 and (R1 – 1) inclusive to see if they are valid. If they are equal, then that single address is checked. Valid addresses are those in logical RAM (0 - 32M) which have memory mapped to them, and also the second mapping of screen RAM at the start of physical memory (32M). From RISC OS 3 onwards, if the area is not recognised as valid Service_ValidateAddress is issued, which if claimed indicates the area is valid, and results in the C flag being cleared on exit from the SWI. See page 1-368.

Related SWIs

None

Related vectors

None

OS_ClaimScreenMemory (SWI &41)

Use spare screen memory

On entry

R0 = 0 for release, 1 for claim
R1 = length required in bytes (if R0 = 1)

On exit

R0 preserved
if the C flag is 0, then memory was claimed successfully
    R1 = length available
    R2 = start address
if the C flag is 1, then memory could not be claimed
    R1 = length that is available

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

There are several restrictions to the use of screen memory. It can only be claimed by one ‘client’ at a time, who gets all of it. It can only be claimed if no bank other than bank 1 has been used. You can’t claim it, for example, if the shadow bank has been used.

While you have claimed the screen memory, you must not perform any action which might causes the screen to scroll. This means avoiding the use of routines which might cause screen output.

It is important to release the memory after it has been used.

This call is mainly intended for internal use; you should not need to use it.

Related SWIs

None

Related vectors

None

OS_ReadRAMFsLimits (SWI &4A)

Get the current limits of the RAM filing system

On entry

—

On exit

R0 = start address
R1 = end address + 1 byte

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 reads the start and end addresses of the RAM filing system. This information can also be read from OS_ReadDynamicArea.

If the RamFS is configured to zero size then R0 and R1 have the same value on exit.

The size of the RamFS after a hard reset (ie the difference between the two return values) can be configured using *Configure RamFsSize.

Related SWIs

OS_ReadDynamicArea (page 1-396)

Related vectors

None

OS_ReadMemMapInfo (SWI &51)

Read the page size and count

On entry

—

On exit

R0 = page size in bytes
R1 = number of pages

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 reads the page size used by MEMC and the number of pages in use. The valid page numbers are 0 to R1 - 1, and the total memory size is R0 × R1 bytes.

Related SWIs

None

Related vectors

None

OS_ReadMemMapEntries (SWI &52)

Read by page number the logical to physical memory mapping used by MEMC

On entry

R0 = pointer to buffer to receive request list

On exit

R0 preserved

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 reads the logical to physical memory mapping used by MEMC. For given page numbers, it finds the corresponding logical address and protection level.

The returned request list is a series of entries three words long, terminated by a –1 in the first word. The three words are used for:

Word	Meaning
1	page number (from 0 upwards)
2	logical address that it is mapped to
3	protection level: 0 readable and writable by everybody 1 read-only in user mode 2 or 3 inaccessible in user mode All other values are reserved.

On entry, the page number fields must be set; on exit, all fields are set.

Related SWIs

OS_SetMemMapEntries (page 1-394), OS_FindMemMapEntries (page 1-398)

Related vectors

None

OS_SetMemMapEntries (SWI &53)

Write the logical to physical memory mapping used by MEMC

On entry

R0 = pointer to request list

On exit

R0 preserved

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 writes the logical to physical memory mapping used by MEMC.

The request list is a series of entries three words long, terminated by a –1 in the first word. The three words are used for:

Word	Meaning
1	page number (from 0 upwards)
2	logical address that it is mapped to
3	protection level: 0 readable and writable by everybody 1 read-only in user mode 2 or 3 inaccessible in user mode All other values are reserved and must not be used.

All fields must be set on entry.

Any address above 32Mbyte (&2000000) makes that page inaccessible. This also sets the protection level to minimum accessibility. For future compatibility, you should use an address of –1 (&FFFFFFFF) for this.

This SWI assumes you know what you are doing. It will set any page to any address, with no checks at all.

If you are using this call, then you can only use OS_ChangeDynamicArea if the kernel's limits are maintained, and all appropriate areas contain continuous memory.

Related SWIs

OS_ChangeDynamicArea (page 1-384), OS_ReadMemMapEntries (page 1-392), OS_FindMemMapEntries (page 1-398)

Related vectors

None

OS_ReadDynamicArea (SWI &5C)

Read the space allocation of a dynamic area

On entry

R0 = area to read

On exit

R0 = pointer to start of area
R1 = current number of bytes in area
R2 = maximum size of area, if bit 7 of R0 was set on entry; preserved otherwise

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 SWI reads the size and – optionally – the maximum size of an area. The area read depends on R0 as follows:

Value of R0	Area to read
0	system heap
1	RMA
2	screen area
3	sprite area
4	font cache
5	RAM filing system

RISC OS 2 ignores bit 7 of R2, and always preserves R2.

Related SWIs

OS_ChangeDynamicArea (page 1-384)

Related vectors

None

OS_FindMemMapEntries (SWI &60)

Read by logical address the logical to physical memory mapping used by MEMC

On entry

R0 = pointer to request list

On exit

R0 preserved

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 reads the logical to physical memory mapping used by MEMC. For a given logical address, it finds the corresponding page number and protection level.

The request list is a series of entries three words long, terminated by a –1 in the first word. The three words are used for:

Word	Meaning
1	page number (from 0 upwards)
2	logical address that it is mapped to
3	protection level: 0 readable and writable by everybody 1 read-only in user mode 2 or 3 inaccessible in user mode All other values are reserved.

On entry, the logical address fields must be set. You may supply probable page numbers, which (if correct) will make this call return more quickly than it might otherwise. If you have no idea what the page number might be, you should set the page number to zero on entry. The protection level is ignored on entry.

If the page number is –1 on exit, then the memory map entry was not found; in this case, the protection level will always be 3. Otherwise the request list has been updated with the page number and protection level for the given logical address.

This call is not available in RISC OS 2.

Related SWIs

OS_ReadMemMapEntries (page 1-392), OS_SetMemMapEntries (page 1-394)

Related vectors

None

*Commands

*Configure

Sets the value of a configuration option in the CMOS RAM

Syntax

```
*Configure [option [value]]
```

Parameters

option	the name of a configuration option
value	its new value(s)

Use

*Configure sets the value of a configuration option in the CMOS RAM. These are used to permanently store the *configuration* (or set-up) of the computer. The time they take effect differs; some take effect immediately, whereas some are only made current on initial power-on or after a hard break (Ctrl-Reset).

If no parameters are specified, the available configuration options are listed.

If parameters are specified, the given value is stored in the location in CMOS RAM appropriate for the given option. Some options require more than one value, and some require none at all.

Where a number is required, you may give it in decimal, as a hex number preceded by &, or a number of the form *base_num*, where *base* is the base of the number in decimal in the range 2 to 36. For example 2_1010 is another way of saying 10.

Here is a list of the available configuration options, the details of which can be found on the appropriate pages:

User Preferences	**Chapter**	**Page**
*Configure Boot	*FileSwitch*	2-149
*Configure Caps	*Character Input*	1-947
*Configure Delay	*Character Input*	1-948
*Configure Dir	*FileCore*	2-254
*Configure DumpFormat	*FileSwitch*	2-150
*Configure FileSystem	*FileSwitch*	2-151
*Configure FontMax1	*The Font Manager*	3-510

User Preferences	Chapter	Page
*Configure FontMax2	*The Font Manager*	3-512
*Configure FontMax3	*The Font Manager*	3-514
*Configure FontMax4	*The Font Manager*	3-516
*Configure FontMax5	*The Font Manager*	3-516
*Configure Language	*The rest of the kernel*	1-978
*Configure Lib	*NetFS*	2-380
*Configure Loud	*VDU Drivers*	1-756
*Configure Mode	*VDU Drivers*	1-757
*Configure MouseStep	*VDU Drivers*	1-761
*Configure NoBoot	*FileSwitch*	2-152
*Configure NoCaps	*Character Input*	1-949
*Configure NoDir	*FileCore*	2-255
*Configure NoScroll	*VDU Drivers*	1-762
*Configure Quiet	*VDU Drivers*	1-763
*Configure Repeat	*Character Input*	1-950
*Configure Scroll	*VDU Drivers*	1-765
*Configure ShCaps	*Character Input*	1-951
*Configure SoundDefault	*The Sound system*	4-59
*Configure Truncate	*FileSwitch*	2-153
*Configure WimpAutoMenuDelay	*The Window Manager*	3-265
*Configure WimpDoubleClickDelay	*The Window Manager*	3-266
*Configure WimpDoubleClickMove	*The Window Manager*	3-267
*Configure WimpDragDelay	*The Window Manager*	3-268
*Configure WimpDragMove	*The Window Manager*	3-269
*Configure WimpFlags	*The Window Manager*	3-270
*Configure WimpMenuDragDelay	*The Window Manager*	3-272
*Configure WimpMode	*The Window Manager*	3-273

Hardware configuration	Chapter	Page
*Configure Baud	*Serial device*	2-483
*Configure Country	*International module*	3-786
*Configure Data	*Serial device*	2-485
*Configure Drive	*ADFS*	2-307
*Configure DST	*The Territory Manager*	3-852
*Configure Floppies	*ADFS*	2-308
*Configure FS	*NetFS*	2-379

Hardware configuration	Chapter	Page
*Configure HardDiscs	*ADFS*	2-309
*Configure IDEDiscs	*ADFS*	2-309
*Configure Ignore	*Character Output*	1-542
*Configure MonitorType	*VDU Drivers*	1-759
*Configure NoDST	*The Territory Manager*	3-853
*Configure Print	*Character Output*	1-544
*Configure PS	*NetPrint*	2-409
*Configure Step	*ADFS*	2-311
*Configure Sync	*VDU Drivers*	1-766
*Configure Territory	*The Territory Manager*	3-854
*Configure TimeZone	*The Territory Manager*	3-855
*Configure TV	*VDU Drivers*	1-767

Memory allocation	Chapter	Page
*Configure ADFSbuffers	*ADFS*	2-305
*Configure ADFSDirCache	*ADFS*	2-306
*Configure FontMax	*The Font Manager*	3-508
*Configure FontSize	*The Font Manager*	3-520
*Configure PrinterBufferSize	*Memory Management*	1-404
*Configure RamFsSize	*RamFS*	2-321
*Configure RMASize	*Memory Management*	1-405
*Configure ScreenSize	*VDU Drivers*	1-764
*Configure SpriteSize	*Sprites*	1-843
*Configure SystemSize	*Memory Management*	1-406d

Example

```
*Configure Baud 7
```

Related commands

*Status

Related SWIs

OS_Byte 162 (page 1-371)

Related vectors

None

*Configure PrinterBufferSize

Sets the configured amount of memory reserved for printer buffering

Syntax

```
*Configure PrinterBufferSize mK|n
```

Parameters

*m*K	number of kilobytes of memory reserved
n	number of pages of memory reserved; n ≤ 127

Use

*Configure PrinterBufferSize sets the configured amount of memory reserved for printer buffering after the next hard reset.

If the parameter is 0, a default amount of memory is reserved.

Example

```
*Configure PrinterBufferSize 32K
```

Related commands

None

Related SWIs

None

Related vectors

None

*Configure RMASize

Sets the configured extra area of memory reserved for relocatable modules

Syntax

```
*Configure RMASize mK|n
```

Parameters

*m*K	number of kilobytes of memory reserved
n	number of pages of memory reserved; n ≤ 127

Use

*Configure RMASize sets the configured extra area of memory reserved in the relocatable module area (RMA) after all modules have been initialised.

If the parameter is 0, no extra memory is reserved.

Example

```
*Configure RMASize 128K
```

Related commands

None

Related SWIs

OS_ChangeDynamicArea (page 1-384)

Related vectors

None

*Configure SystemSize

Sets the configured extra area of memory reserved for the system heap

Syntax

```
*Configure SystemSize mK|n
```

Parameters

*m*K	number of kilobytes of memory reserved
n	number of pages of memory reserved; n ≤ 63

Use

*Configure SystemSize sets the configured extra area of memory reserved for the system heap after all modules have been initialised.

If the parameter is 0, no extra memory is reserved.

Example

```
*Configure SystemSize 32K
```

Related commands

None

Related SWIs

OS_ChangeDynamicArea (page 1-384)

Related vectors

None

*Status

Provides information on how the computer is configured

Syntax

`*Status [`*`option`*`]`

Parameters

`option`	the name of a configuration option

Use

*Status displays the value of a configuration option in the CMOS RAM. If no option is specified, the values of all configuration options are shown.

Because the values of these configuration options are held in non-volatile memory (the battery-backed CMOS RAM) they are preserved even when the computer is switched off, until reset by using either the Configure application from the desktop or the *Configure command from the command line.

Example

```
*Status TV
```

Related commands

*Configure

Related SWIs

OS_Byte 161 (page 1-369), OS_ReadDynamicArea (page 1-396)

Related vectors

None

*Status

17 Time and Date

Introduction

There are two basic aspects of time dealt with in this chapter: passive aspects such as reading various clock settings; and active ones, where an event occurs when a given time is reached. In this chapter, a *clock* is a place where a stored value is incremented on a regular basis. The *time* is the name of the value as it is read or written.

There are several clocks that increment every 1/100th of a second (centisecond). One of them cannot be changed except by a hard reset. This is useful for time-stamping events, such as mouse moves. Another can be changed by a program, so is useful for elapsed time calculations.

The real-time clock keeps the real-world time, and represents time in centiseconds since 00:00:00 on January 1 1900. There are calls to present this information in a number of ways. The real-time can be converted to a string with complete program control over its format.

A variety of timer events can be set up. There are SWIs that will call your application after a given delay has passed or every time that delay has elapsed. You can set up a routine to sit on the ticker vector, to enable it to be called every centisecond.

A specialised form of timer event is one that will occur every time the screen driving hardware reaches the bottom of the screen. This event is useful for flicker-free redrawing. See the chapter entitled *VDU Drivers* on page 1-547 for further details.

Overview and Technical Details

There are four timers, which increment at a centisecond rate. They are:

- the monotonic timer (read-only)
- the system timer (read/write)
- the interval time (read/write)
- the real-time clock (read only in general – ie only users should change it).

Monotonic timer

A monotonic timer cannot be written, except by a hard reset or when the machine is turned on. OS_ReadMonotonicTime (page 1-446) allows you to read this value. It is useful for time-stamping within an application, such as event times. Because it can never be changed, the order of events cannot be confused.

It is stored as a 4-byte value with least significant byte first. It is incremented every centisecond, which means that it would take nearly 500 days for it to wrap around.

System clock

The system clock is stored as a 5-byte value. Like the monotonic timer it is reset by hard resets and increments every centisecond. However it can be altered. This is useful for measuring elapsed times in an application. OS_Word 1 reads the value and OS_Word 2 writes it.

Real-time

The real-time clock is stored as a 5-byte value in the CMOS clock chip and reflects the normal usage of the word clock. That is, it stores the elapsed number of centiseconds since 00:00:00 on January 1 1900. You can set it using the Alarm application on the desktop.

Under RISC OS 2 the real-time clock is assumed to be set to local time. Under later versions, the real-time clock is assumed to be set to UTC, or *Universal Time Coordinated*. (This used to be called GMT, or *Greenwich Mean Time*.) Territory modules provide the necessary information for the kernel to convert the real-time clock value to a local time in a suitable format.

A soft copy of the real time clock is also kept by RISC OS and is used by the filing system to date-stamp files. This soft copy is updated from the CMOS clock chip following a hard reset.

String format

*Time displays the local time and date as a string. It calls OS_Word 14,0 to do so. The format of the string depends on the territory which the computer is set to use. (It is fixed in RISC OS 2, which does not support territories.) For example:

```
Tue,28 Mar 1989.13:25:54
```

5-byte format

The real-time clock can be read in the standard 5-byte format using OS_Word 14,3. This, or any, 5-byte time can be converted into a string using OS_ConvertStandardDateAndTime (page 1-449) or OS_ConvertDateAndTime (page 1-447).

Changing real-time

The real-time clock's time of day can be altered with OS_Word 15,8, its date with OS_Word 15,15, or both with OS_Word 15,24. These calls all use the time in a string format (see above).

Format field names

For most of the above calls the time string is passed or returned in a fixed, call-dependent format. However, for some calls you can customise the way that the time and date is presented by supplying a format string. The string is copied character for character to the output buffer unless a '%' is found. If this character is followed by any of the following codes – which may be in upper or lower case – then the appropriate value is copied to the output buffer:

Name	Value	Examples (UK)
CS	Centiseconds	99
SE	Seconds	59
MI	Minutes	05
12	Hours in 12 hour format	07
24	Hours in 24 hour format	23
AM	AM or PM indicator (in local language)	PM
PM	AM or PM indicator (in local language)	AM
WE	Weekday – full (in local language)	Thursday
W3	Weekday – short (in local language)	Thu
WN	Weekday – number	5
DY	Day of the month (in local language)	01
ST	Ordinal pre/suffix (in local language)	st nd rd th
MO	Month name – full (in local language)	September
M3	Month name – short (in local language)	Sep
MN	Month – number	09
CE	Century	19
YR	Year within century	87
WK	Week of year (using local start of week)	52
DN	Day of the year	364
TZ	Timezone	BST
0	Insert an ASCII 0 zero byte	
%	Insert a '%'	

You must not make **any** assumptions about the nature or length of any fields that use the local language. For example: short forms of the weekday or month are three characters long in the UK territory, but may have a different length in other territories; the day of the month may not be numeric; ordinals may be null; and so on. However, you can find out the maximum length of fields for your territory by calling Territory_ReadCalendarInformation (page 3-847).

To cause leading zeros to be omitted, prefix the field with the letter 'Z'. For example, '%zmn' means the month number without leading zeros. '%0' may be used to split the output into several zero-terminated strings.

As an example, this format string:

```
%W3,%DY %M3 %CE%YR.%24:%MI:%SE
```

would produce this time string in the UK territory:

```
Tue,28 Mar 1989.13:25:54
```

OS_ConvertDateAndTime (page 1-449) will convert a 5-byte time into a string using a supplied format string.

BCD conversions

The CMOS clock chip stores the time internally in a Binary Coded Decimal (BCD) format. OS_Word 14,1 will read the time as a 7-byte BCD block. OS_Word 14,2 will convert this BCD block into a string. These calls are provided for compatibility only, and you should not use them.

Timer events

There are three different causes of timer events: the interval timer, the timer chain and the VSync timer. You should not use these under the Wimp, as you cannot guarantee that your task will be paged in at the time of the event.

Interval timer

The interval timer is a 5-byte clock that increments every centisecond. If enabled by OS_Byte 14, an event will occur when the counter reaches zero. Thus to wait for a given time, the interval timer must be set to the negative of it using OS_Word 4. OS_Word 3 can read the current setting of the interval timer.

For example, to wait 10 seconds, –1000 must be passed to OS_Word 4.

The interval timer is kept for compatibility with earlier Acorn operating systems. Its use should be avoided if possible. It is especially important that this is not used under the Wimp, since it cannot cope with more than one program using it at once.

Timer chain

An easier to use and more sophisticated way for an application to be called at a given time is the timer chain. These are independent of event routines, but are used in a similar manner. OS_CallAfter (page 1-441) can be used to get a given address to be called after a certain time has elapsed. OS_CallEvery (page 1-443) is like this, but automatically reloads the counter when it has expired. OS_RemoveTickerEvent (page 1-445) will cancel either OS_CallAfter before it occurs or OS_CallEvery to stop it repeating forever.

OS_CallAfter and OS_CallEvery are passed an address to call, the delay to wait and an identification word to return in R12. Thus, many timers can be running concurrently.

These are stored in a list which can be any size up to the machine memory limit.

Vertical sync timer

The screen is refreshed at a mode-dependent rate: typically from 50Hz (standard monitor type modes) to 70Hz or more (particularly with VGA or Super VGA monitor modes). From the time that the bottom of the screen is complete till the top of the screen commences again is a delay called the Vertical sync period. This allows the electron beam to go to this start position. The Vertical sync event coincides with the vertical sync beginning. You can use OS_Byte 14 to enable this event, so that flicker-free re-drawing can be done while the VDU is not being written to.

OS_Byte 176 provides access to a one byte counter that decrements at the rate of the Vertical sync event. Because the rate of this timer varies, you should not use it for running timing loops for games, music, etc.

Obsolete timers

OS_Byte 243 reads a temporary location used by the timer software. It is kept for compatibility with earlier Acorn operating systems and must not be used.

SWI Calls

OS_Byte 176 (SWI &06)

Read/write 50Hz counter

On entry

R0 = 176
R1 = 0 to read or new 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 reads or writes a one-byte counter which is decremented at the rate of the Vertical sync interrupt. This rate is mode and monitor dependent, typically in the range 50 - 70Hz, Consequently you should not use this timer for precise timing loops.

Related SWIs

None

Related vectors

ByteV

OS_Byte 243 (SWI &06)

Read timer switch state

On entry

R0 = 243
R1 = 0
R2 = 255

On exit

R0 preserved
R1 = switch state
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

In order to protect the centisecond clock against corruption during reset, the OS keeps two copies. One of them is the one which will be read or written when one of the OS_Words is called, the other is the one which will be updated during the next 100Hz interrupt. When the update has been performed correctly, the values are swapped. This OS_Byte enables you to read the byte which indicates which copy is being used. Its only practical use is as a location which changes 100 times a second.

This call is obsolete and should not be used.

Related SWIs

OS_Word 3 (page 1-423), OS_Word 4 (page 1-425)

Related vectors

ByteV

OS_Word 1 (SWI &07)

Read system clock

On entry

R0 = 1
R1 = pointer to five byte block

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains the value of the system clock at the instant of the call.

R1+0 = time (least significant byte)
R1+1 = …
R1+2 = …
R1+3 = …
R1+4 = time (most significant byte)

The clock is incremented every centisecond. The value of the clock is preserved over a soft break and set to zero after a hard break.

Related SWIs

OS_Word 2 (page 1-421)

Related vectors

WordV

OS_Word 2 (SWI &07)

Write system clock

On entry

R0 = 2
R1 = pointer to five byte block with centisecond clock value in it

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On entry, the parameter block contains the value to set the system clock.

R1+0 = time (least significant byte)
R1+1 = …
R1+2 = …
R1+3 = …
R1+4 = time (most significant byte)

This allows the clock to be set to a specified value.

Related SWIs

OS_Word 1 (page 1-419)

Related vectors

WordV

OS_Word 3 (SWI &07)

Read interval timer

On entry

R0 = 3
R1 = pointer to five byte block

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains the value of the interval timer at the instant of the call.

R1+0 = time (least significant byte)
R1+1 = …
R1+2 = …
R1+3 = …
R1+4 = time (most significant byte)

Related SWIs

OS_Word 4 (page 1-425)

Related vectors

WordV

OS_Word 4 (SWI &07)

Write interval timer

On entry

R0 = 4
R1 = pointer to five byte block

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On entry, the parameter block contains the value to set the interval timer.

R1+0 = time (least significant byte)
R1+1 = …
R1+2 = …
R1+3 = …
R1+4 = time (most significant byte)

This call resets the interval timer to a specified value.

Like the system clock, the interval timer is incremented 100 times a second. The interval timer can be made to cause an event when its value reaches zero. To do this, it must be set to minus the number of centiseconds that are to elapse before the event takes place.

To produce repeated events, the routine servicing the timer event should reload the timer with the appropriate number. For example, to produce an event every 10 seconds, reload it with –1000 (&FFFFFFFC18). An alternative is to use the special ticker event, described in the chapter entitled *Events* on page 1-147.

Note that you must use OS_Byte 14 to enable the interval timer event.

Related SWIs

OS_Word 3 (page 1-423)

Related vectors

WordV

OS_Word 14,0 (SWI &07)

Read soft copy of the real-time clock as a string, converting to local time

On entry

R0 = 14
R1 = pointer to parameter block
R1+0 = 0 (reason code)

On exit

R0, R1 preserved

Interrupts

Interrupts are enabled (in RISC OS 2, the interrupt status is not altered)
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains the local time as a string terminated by a Return character (ASCII 13). The format of the string depends on the territory which the computer is set to use. (It is fixed in RISC OS 2, which does not support territories.)

This time string comes from the soft copy of the real-time clock maintained by RISC OS, rather than from the CMOS clock chip itself.

This call is equivalent to the *Time command.

Related SWIs

OS_Word 15 (page 1-435)

Related vectors

WordV

OS_Word 14,1 (SWI &07)

Read time in Binary Coded Decimal (BCD) format, converting to local time

On entry

R0 = 14
R1 = pointer to parameter block
R1+0 = 1 (reason code)

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains a seven-byte BCD clock value:

R1+0 = year	(00 - 99)
R1+1 = month	(01 - 12; 01 = January etc)
R1+2 = day of month	(01 - 31)
R1+3 = day of week	(01 - 07; 01 = Sunday etc)
R1+4 = hours	(00 - 23)
R1+5 = minutes	(00 - 59)
R1+6 = seconds	(00 - 59)

Under RISC OS 2 the clock value is read directly from the CMOS real time clock chip, which is assumed to be set to local time, and so the value is not further converted. Under later versions of RISC OS the clock is read from a soft copy of the real time, which is assumed to be set to UTC, and so the value is then converted to local time.

Related SWIs

OS_Word 15 (page 1-435)

Related vectors

WordV

OS_Word 14,2 (SWI &07)

Convert BCD clock value into string format

On entry

R0 = 14
R1 = pointer to parameter block

R1+0 = 2	reason code
R1+1 = year	(00 - 99)
R1+2 = month	(01 - 12; 01 = January etc)
R1+3 = day of month	(01 - 31)
R1+4 = day of week	(01 - 07; 01 = Sunday etc)
R1+5 = hours	(00 - 23)
R1+6 = minutes	(00 - 59)
R1+7 = seconds	(00 - 59)

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On entry, the parameter block contains a 7-byte BCD clock value:

On exit, the parameter block contains a string terminated by a Return character (ASCII 13), representing the same time. The format of the string depends on the territory which the computer is set to use. (It is fixed in RISC OS 2, which does not support territories.)

Related SWIs

OS_Word 15 (page 1-435)

Related vectors

WordV

OS_Word 14,3 (SWI &07)

Read real-time in 5-byte format

On entry

R0 = 14
R1 = pointer to parameter block
    R1+0 = 3 (reason code)

On exit

R0 preserved
R1 preserved:
    R1+0 = LSB of time
    R1+1 = …
    R1+2 = …
    R1+3 = …
    R1+4 = MSB of time

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit the parameter block contains the 5-byte real time read directly from the soft copy of the real-time clock. This number gives the elapsed number of centiseconds since 00:00:00 on January 1 1900. Under RISC OS 2 the real-time clock is assumed to be set to local time; under later versions the real-time clock is assumed to be set to UTC.

This 5-byte real-time is used for time/date stamping by the filing system. It is also useful for utilities which are used for building consistent systems, eg ‘Make’.

Related SWIs

OS_Word 15 (page 1-435)

Related vectors

WordV

OS_Word 15,8 (SWI &07)

Writes the time of day to both the CMOS clock and its soft copy

On entry

R0 = 15
R1 = pointer to parameter block
    R1+0 = 8 (reason code)
    R1+1… = string giving time of day (in local language)

On exit

R0, 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 writes the time of day to both the CMOS clock and its soft copy.

On entry, the parameter block contains the local time of day as a string; the format this string must have depends on the territory which the computer is set to use. (It is fixed in RISC OS 2, which does not support territories.)

The format for the UK territory (and for RISC OS 2) is:

HH:*MM*:*SS* *eg* 17:35:04

Related SWIs

OS_Word 14 (page 1-427)

Related vectors

WordV

OS_Word 15,15 (SWI &07)

Writes the date to both the CMOS clock and its soft copy

On entry

R0 = 15
R1 = pointer to parameter block
    R1+0 = 15 (reason code)
    R1+1… = string giving date (in local language)

On exit

R0, R1 preserved

The C flag will be set on exit, if the parameter block contained a format error

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call writes the date to both the CMOS clock and its soft copy.

On entry, the parameter block contains the local date as a string; the format this string must have depends on the territory which the computer is set to use. (It is fixed in RISC OS 2, which does not support territories.)

The format for the UK territory (and for RISC OS 2) is:

Day,*DD MMM YYYY* *eg* Mon,17 Feb 1992

Related SWIs

OS_Word 14 (page 1-427)

Related vectors

WordV

OS_Word 15,24 (SWI &07)

Writes the time of day and date to both the CMOS clock and its soft copy

On entry

R0 = 15
R1 = pointer to parameter block
    R1+0 = 24 (reason code)
    R1+1… = string giving time of day and date (in local language)

On exit

R0, R1 preserved

The C flag will be set on exit, if the parameter block contained a format error.

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call writes the time of day and date to both the CMOS clock and its soft copy.

On entry, the parameter block contains the local time of day and date as a string; the format this string must have depends on the territory which the computer is set to use. (It is fixed in RISC OS 2, which does not support territories.)

The format for the UK territory (and for RISC OS 2) is:

Day,*DD MMM YYYY*.*HH*:*MM*:*SS* *eg* `Mon,17 Feb 1992.17:35:04`

Related SWIs

OS_Word 14 (page 1-427)

Related vectors

WordV

OS_CallAfter (SWI &3B)

Call a specified address after a delay

On entry

R0 = time in centiseconds
R1 = address to call
R2 = value of R12 to call code with

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_CallAfter calls the code pointed to by R1 after the delay specified in R0. The code is called in SVC mode with interrupts disabled. It must preserve all registers, and return using the instruction `MOV R15, R14`.

OS_RemoveTickerEvent can be used to cancel a pending OS_CallAfter

In RISC OS 2 this call may return incorrect error pointers. An invalid value of R0 now generates the error message ‘Invalid time interval’, rather than the null string generated by RISC OS 2.

Related SWIs

OS_CallEvery (page 1-443), OS_RemoveTickerEvent (page 1-445)

Related vectors

None

OS_CallEvery (SWI &3C)

Call a specified address every time a delay elapses

On entry

R0 = (delay in centiseconds) – 1
R1 = address to call
R2 = value of R12 to call code with

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_CallEvery calls the code pointed to by R1 every (R0+1) centiseconds, until OS_RemoveTickerEvent is executed or Break is pressed. The code is called in SVC mode with interrupts disabled. It must preserve all registers, and return using the instruction `MOV R15, R14`.

The minimum value for R0 is 1, which means the minimum possible delay is 2 centiseconds. If you wish to be called every centisecond, you must instead claim TickerV.

In RISC OS 2 this call may return incorrect error pointers. An invalid value of R0 now generates the error message ‘Invalid time interval’, rather than the null string generated by RISC OS 2.

Related SWIs

OS_CallAfter (page 1-441), OS_RemoveTickerEvent (page 1-445)

Related vectors

None

OS_RemoveTickerEvent (SWI &3D)

Remove a given call address and R12 value from the ticker event list

On entry

R0 = call address
R1 = value of R12 used in OS_CallEvery or OS_CallAfter

On exit

R0, R1 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_RemoveTickerEvent takes R0 as the address and R1 as the R12 value of the event to find and remove from its list.

It is used to stop an event set up by a call to OS_CallAfter or OS_CallEvery. The parameters passed must match those originally passed to OS_CallEvery or OS_CallAfter for it to remove the correct event.

Related SWIs

OS_CallAfter (page 1-441), OS_CallEvery (page 1-443)

Related vectors

None

OS_ReadMonotonicTime (SWI &42)

Number of centiseconds since the last hard reset

On entry

—

On exit

R0 = time in centiseconds

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadMonotonicTime returns the number of centiseconds since the last hard reset, or switching on of the machine. ‘Monotonic’ refers to the fact that this timer is guaranteed to change with time (increasing until it wraps around). It is used, for example, to time-stamp mouse events.

Related SWIs

None

Related vectors

None

OS_ConvertStandardDateAndTime (SWI &C0)

Converts a 5-byte time into a string

On entry

R0 = pointer to 5-byte time block
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating zero in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ConvertStandardDateAndTime converts a five-byte value representing the number of centiseconds since 00:00:00 on January 1st 1900 into a string. It converts it using a standard format string stored in the system variable ‘Sys$DateFormat’ and places it in a buffer (which should be at least 20 bytes).

For details of the format field names see the section entitled *Format field names* on page 1-412.

From RISC OS 3 onwards this SWI simply calls Territory_ConvertStandardDateAndTime, which you should use instead.

Related SWIs

OS_ConvertDateAndTime (page 1-449),
Territory_ConvertStandardDateAndTime (page 3-817)

Related vectors

None

OS_ConvertDateAndTime (SWI &C1)

Convert 5-byte time into a string using a supplied format string

On entry

R0 = pointer to 5-byte time block
R1 = pointer to buffer for resulting string
R2 = size of buffer
R3 = pointer to format string (null terminated)

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating zero in buffer
R2 = number of free bytes in buffer
R3 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ConvertDateAndTime converts a five byte value representing the number of centiseconds since 00:00:00 on January 1st 1900 into a string. It converts it using the format string supplied.

Apart from the following exception, the format string is copied directly into the result buffer. However, whenever '%' appears in the format string, the next one or two characters are treated as a special field name which is replaced by a component of the current time.

For details of the format field names see the section entitled *Format field names* on page 1-412.

From RISC OS 3 onwards this SWI simply calls Territory_ConvertDateAndTime, which you should use instead.

Related SWIs

OS_ConvertStandardDateAndTime (page 1-447), Territory_ConvertDateAndTime (page 3-817)

Related vectors

None

*Commands

*Time

Displays the day, date and time of day

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 14,0 (page 1-427), OS_Word 15 (page 1-435), OS_ConvertStandardDateAndTime (page 1-447), OS_ConvertDateAndTime (page 1-449), Territory_ConvertDateAndTime (page 3-815), Territory_ConvertStandardDateAndTime (page 3-817)

Related vectors

WordV, WrchV

18 Conversions

Introduction

This chapter is a collection of SWIs that convert data from one form to another. Here is a summary of the conversions that can be done:

- Convert a number to a string in binary, decimal or hex, with some format control. You can specify the source number in a variety of sizes: 1, 2, 3 or 4 bytes in length in most cases.
- Convert a string containing a number in any base from 2 to 36 to a number.
- Process a string with control codes and other special characters. This allows a string with any control codes to be created by passing a string with only printable characters in it.
- Substitute a string containing arguments with the given values. Used with command line arguments to an application.
- Evaluate an expression with logical, arithmetic, bit and string operations, giving a logical, numeric or string result.
- Extract options from a command line using a given key.
- Convert a SWI number to a string with its full name and vice versa.
- Convert a network station pair of numbers number into a string.
- Convert a file size into a string, for example ‘12 Kbytes’

Overview and Technical Details

This section leads through the details of the differing conversion calls. Whilst most are mutually independent, some SWIs may use others within this chapter to give a multi-layered functionality.

Numbers to strings

The simplest option to convert a signed 32-bit integer into a string, the most common operation, is to use OS_BinaryToDecimal (page 1-468).

For a far greater functionality, there is a set of 24 SWIs with a common calling convention that allow a wide ranging list of conversions. Generically, these SWIs are called OS_Convert*NameNumber* (page 1-482). The *Name* refers to the destination format of the string. It can be hex, signed and unsigned integer (optionally with spaces between the thousands, millions and so on), or binary. The *Number* is the number of bytes to use on input. For all apart from hex, this is 1, 2, 3, or 4 bytes. Hex can be 1, 2, 4, or 8 nibbles long. See the description of these SWIs for detail.

Note that OS_BinaryToDecimal is equivalent to OS_ConvertInteger4 (page 1-483) from these SWIs.

Strings to numbers

OS_ReadUnsigned (page 1-460) will read a number in an ASCII string and convert it into an unsigned integer. The number in the string can be specified to be in any base from 2 to 36. Base 36 has 0 - 9, A - Z as numbers. No prefix means that the number is decimal by default, while the conventional ‘&’ is used to indicate hex. All bases can be specified by the *base_number* form: eg 2_1100 is 12 in binary.

GS string operations

The GS operations are a way of putting any characters from 0 - 255 into a string using only the printable character set. OS_GSInit (page 1-462) and OS_GSRead (page 1-464) work together to scan a string on a character at a time basis. OS_GSTrans (page 1-466) performs both these functions and scans the string. Unless you need character by character control, OS_GSTrans is easier to use.

| character

The ‘|’ character is used by OS_GSRead and OS_GSTrans as a flag for a special character. It affects how the character following it is interpreted. Here is a list of its effects:

ASCII code	Symbols used
0	|@
1 - 26	|*letter* eg |A (or |a) = ASCII 1, |M (or |m) = ASCII 13
27	|[or |{
28	|\
29	|] or |}
30	|^ or |~
31	|_ or |’ (grave accent)
32 - 126	*keyboard character*, except for:
"	|"
|	||
<	|<
127	|?
128 - 255	|!*coded symbol* eg ASCII 128 = |!|@ ASCII 129 = |!|A

Note that ‘|!’ means set the top bit of the following character, even if it is set by another ‘|’ character.

To include leading spaces in a definition, the string must be in quotation marks, ‘"’, which are not included in the definition. To include a single " character in the string, use |" or "".

Substitute arguments

The reason why ‘<’ must be preceded by a ‘|’ is that you can put values and variables inside angle brackets.

You can use the form <*number*>, where the number between the angle brackets will be interpreted as if it was a parameter to OS_ReadUnsigned: that is, a number in any base from 2 to 36. The value returned from this SWI will be placed as a character in the output stream; bits 8 - 31 are ignored.

A string with a name enclosed in ‘< >’ characters will be used to look up a system variable. You must have used *Set, *SetMacro or *SetEval to set the variable. The value of the variable will be substituted for the name and the angle brackets using OS_ReadVarVal; eg if the variable ‘hisname’ had been set to ‘Fred’, then the string ‘My friend’s name is <hisname>’ would be translated to ‘My friend’s name is Fred’. System variables and the calls that operate on them are described in the chapter entitled *Program Environment* on page 1-287.

Flags

There are options which can be used to determine the way in which the string is interpreted. This is done by setting the top three bits in R2 passed to OS_GSInit or OS_GSTrans, as follows:

Bit	Meaning
29	If set then a space is treated as a string terminator
30	If set control codes are not converted (ie '|' syntax is ignored)
31	Double quotation marks '"' are not to be treated specially, ie they are not stripped around strings.

*Echo

The *Echo command will pass a string through OS_GSTrans and then send it to the display.

Evaluation operators

A string containing an expression can be evaluated. An expression consists of any of the operators listed below, brackets (for grouping), strings, and numbers. It can return a result that is a number or a string. OS_EvaluateExpression (page 1-470) is the core routine here. It is in turn called by *Eval. This allows you to perform evaluations from the command line.

The *If command also uses this call to perform a logical decision about which
* Command to perform.

Any strings in the evaluation string are passed to OS_GSTrans, so all its operators will be used. This of course means that OS_ReadUnsigned and OS_ReadVarVal will in turn be called if you use a string that requires them. Note, however, that vertical bar escape sequences (eg '|G' for ASCII 7) are not recognised.

As well as passing <*name*> operators in strings to OS_ReadVarVal, any item which cannot immediately be treated as a string or a number is also looked up as a system variable. For example, in the expression FRED+1, FRED will be looked up as a variable.

The operators recognised by the expression evaluator are as follows:

Arithmetic operators

+	Add two integers
–	Subtract two integers
*	Multiply two integers
/	Integer part of division
MOD	Remainder of a division

Logical operators

=	Equal	–1 is TRUE
<>	Not equal	0 is FALSE
>=	Greater than or equal	
<=	Less than or equal	
<	Less than	
>	Greater than	

Bit operators

>>	Arithmetic shift right
>>>	Logical shift right
<<	Logical shift left
AND	AND
OR	OR
EOR	Exclusive OR
NOT	NOT

String operators

+	Concatenate two strings	eg "HI" + "LO" = "HILO"
RIGHT *n*	Take *n* characters from the right eg "HELLO" RIGHT 2 = "LO"	
LEFT *n*	Take *n* characters from the left eg "HELLO" LEFT 3 = "HEL"	
LEN	Return the length of a string	eg LEN "HELLO" = 5

Conversions

STR	Convert a number into a string	eg STR 24 = "24"
VAL	Take the value of a string	eg VAL "12d3" = 12

Where appropriate, type conversions are performed automatically. For example, if an integer is subtracted from a string, then the string is evaluated and an integer result is produced ("2"–1 gives the result 1). The null string "" is converted to 0 by both the implicit and explicit (VAL) conversions.

Similarly, integers will be converted to strings if necessary: the expression 1234 LEFT 2 will yield "12".

The operators have the same relative priorities as their equivalents in BBC BASIC, eg * is higher than + which is higher than >, etc. Remember you can use brackets to override this standard precedence.

Parameter substitution

Given a list of space separated arguments, OS_SubstituteArgs (page 1-476) will replace references to those parameters in a string: %0 refers to the first string in the argument list and so on. This is generally used when processing command lines.

For a more powerful handling of command lines, use OS_ReadArgs (page 1-478). This is passed a list of parameter definitions and an input string. The parameters can be described as being in any order or in a fixed order. They can handle on/off switches (ie presence is indicated), or values. The values can also be automatically passed through OS_GSTrans or OS_EvaluateExpression if required.

SWI number to/from string

Two calls can be used to translate a SWI number to and from its full name as a string. OS_SWINumberToString (page 1-472) will convert from a SWI number to a string, and OS_SWINumberFromString (page 1-474) will convert from a string to a SWI number.

Note that having bit 17 set will result in the string being prefixed with an ‘X’, and vice versa.

Econet numbers

The pair of numbers that refer to the network number and station number can be converted into a string by OS_ConvertFixedNetStation (page 1-486). This will pad the string with leading zeros where required. If you don’t want this padding, use OS_ConvertNetStation (page 1-488).

File size

There are two SWIs that will convert a file size from an integer into a string. They can decide whether to display as bytes, Kbytes or Mbytes. OS_ConvertFileSize (page 1-492) will convert an integer into a number up to 4 digits followed by an optional ‘K’ if it is in kilobytes or ‘M’ if in megabytes, followed by the word ‘bytes’ and a null to terminate.

OS_ConvertFixedFileSize (page 1-490) is exactly the same, except that it will always print the numeric field as four characters, padding with spaces if necessary.

SWI Calls

OS_ReadUnsigned (SWI &21)

Convert a string to an unsigned number

On entry

R0 = base in the range 2 - 36 (else 10 assumed), and flags in top 3 bits
R1 = pointer to string
R2 = maximum value if R0 bit 29 set

On exit

R0 preserved
R1 = pointer to terminator character
R2 = value

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadUnsigned takes a pointer to a string and tries to convert it into an integer value which is returned in R2.

Valid strings may start with a digit (where ‘digits’ may also be letters, depending on the base) or one of the following:

&	The number is in hexadecimal notation
*base*_	The number is in a given base, where *base* is in the range 2 to 36. For example, 2_1010 is a base two (binary) number.

These override any base specified in R0. (If R0 contains an illegal base, 10 is assumed.) Characters following them are read until a character is reached which is not consistent with the base in use. For example, assuming R0=10 on entry, the terminator of 43AZ is A, whereas the terminator of &43AZ is Z.

In addition, R0 contains three flags which cause checks to be performed on the terminator and the range of the number obtained:

Bit	Meaning if set
29	Restrict range to 0 - R2 inclusive; a ‘Number too big’ error is given otherwise
30	Restrict value range to 0 - 255
31	Check terminator is a control character or space

If either of these checks fail, a ‘Bad number’ error is given. This error also occurs if the first character is not a valid digit. If a base is given at the start of the number and isn’t in the range 2 - 36, a ‘Bad base’ error is given.

Related SWIs

None

Related vectors

None

OS_GSInit (SWI &25)

Initialises registers for use by OS_GSRead

On entry

R0 = pointer to string, terminated by ASCII 10 (LF) or 13 (CR) or 0 (NUL)
R2 = flags

On exit

R0 = value to pass back in to OS_GSRead
R1 = first non-blank character
R2 = value to pass back in to OS_GSRead

Interrupts

Interrupt state is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_GSInit is one of the string routines which are used by the operating system command line interpreter to process the strings sent to it. One of the advantages of these routines is that they enable you to use the character '|' to introduce control characters which would otherwise be difficult to enter directly from the keyboard.

See the section entitled *GS string operations* on page 1-454 for a list of the conversions that are performed by the routines, and of the flags passed in R2.

OS_GSInit also returns the first non-blank character in the string. However, this is not necessarily the same as the output from the first OS_GSRead, since OS_GSInit doesn't perform any expansion.

Related SWIs

OS_GSRead (page 1-464), OS_GSTrans (page 1-466)

Related vectors

None

OS_GSRead (SWI &26)

Returns a character from a string which has been initialised by OS_GSInit

On entry

R0 from last OS_GSRead/OS_GSInit
R2 from last OS_GSRead/OS_GSInit

On exit

R0 updated for next call to OS_GSRead
R1 = next translated character
R2 updated for next call to OS_GSRead
C flag is set if end of string reached

Interrupts

Interrupt state is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_GSRead reads a character from a string, using registers initialised by a OS_GSInit immediately prior to this call. The next expanded character is returned in R1. The values in R0 and R2 are updated so they are set up for the next call to OS_GSRead.

The interpretation of characters which pass through OS_GSRead is described in the section entitled *GS string operations* on page 1-454. Note that this call does not correctly handle quoted termination in RISC OS 2.

An error is returned for a bad string – for example, mismatched quotation marks.

Related SWIs

OS_GSInit (page 1-462), OS_GSTrans (page 1-466)

Related vectors

None

OS_GSTrans (SWI &27)

Equivalent to a call to OS_GSInit and repeated calls to OS_GSRead

On entry

R0 = pointer to string, terminated by ASCII 10 (LF) or 13 (CR) or 0 (NUL)
R1 = buffer pointer
R2 = buffer size (*maxlen*) and flags in top 3 bits

On exit

R0 = pointer to character after terminator
R1 = pointer to buffer, or 0
R2 = number of characters in buffer, or *maxlen* if the buffer overflowed
C flag is set if buffer overflowed

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_GSTrans is equivalent to a call to OS_GSInit followed by repeated calls to OS_GSRead until the end of the source string is reached. Each time it obtains a character and translates it, OS_GSTrans then places it in a buffer.

The flags in R2, on entry, are the same as those supplied to OS_GSInit. On exit, R0 points to the character after the terminator of the source string, and R1+R2 points to the terminator of the translated string. If the C flag is set on exit the buffer was too small for the translated string; R2 is set to the length of the buffer.

The flags and interpretation of characters which pass through OS_GSTrans are described in the section entitled *GS string operations* on page 1-454. Note that this call does not correctly handle quoted termination in RISC OS 2.

An error is returned for a bad string – for example, mismatched quotation marks.

Related SWIs

OS_GSInit (page 1-462), OS_GSRead (page 1-464)

Related vectors

None

OS_BinaryToDecimal (SWI &28)

Convert a signed number to a string

On entry

R0 = signed 32-bit integer
R1 = pointer to buffer
R2 = maximum length

On exit

R0, R1 preserved
R2 = number of characters given

Interrupts

Interrupt state is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_BinaryToDecimal takes a signed 32-bit integer in R0 and converts it to a string, placing it in the buffer. R1 points to the buffer and R2 contains its maximum length. Leading zeros are suppressed and the string will start with a minus sign, '–', if R0 was negative. The number of characters given is returned in R2.

The error 'Buffer overflow' is given if the converted string is too long to fit in the buffer.

Related SWIs

None

Related vectors

None

OS_EvaluateExpression (SWI &2D)

Evaluate a string expression and return an integer or string result

On entry

R0 = pointer to string
R1 = pointer to buffer
R2 = length of buffer

On exit

R0 preserved
R1 = 0 if an integer returned, else preserved
R2 = integer result, or length of string in buffer

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_EvaluateExpression takes a string pointed to by R0, evaluates it, and returns the result. The location and type of the result is given by R1 as follows:

Value	Meaning
0	Integer result returned in R2
Not 0	String is returned in buffer pointed to by R1, length returned in R2, R0 and R1 preserved

See the section entitled *Evaluation operators* on page 1-456 for a description of the operators that you can use. Note that monadic plus/minus operators are not correctly handled in RISC OS 2 (eg *Eval 50*–3 gives a ‘Missing operand’ error).

The resulting string is unterminated. If the buffer is not large enough to hold it, then a ‘Buffer overflow’ error is generated.

Related SWIs

None

Related vectors

None

OS_SWINumberToString (SWI &38)

Convert a SWI number to a string containing its name

On entry

R0 = SWI number
R1 = pointer to buffer
R2 = buffer length

On exit

R0, R1 preserved
R2 = length of string in buffer

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_SWINumberToString converts a SWI number to a SWI name.

The returned string is null-terminated, and starts with an X if the SWI number has bit 17 set.

SWI numbers < &200 have an ‘OS_’ prefix to the main part, and a SWI-dependent end section (which is ‘Undefined’ for unknown OS SWIs).

SWI numbers in the range &100 to &1FF are converted in the form OS_WriteI+"A", or OS_WriteI+23 if the character is not a printable one.

SWI numbers Š &200 are looked for in modules. If a suitable name is found, it is given in the form *module_name* or *module_number*, eg. Wimp_Initialise, Wimp_32. If no name is found in the modules, the string ‘User’ is returned.

Note that this call does not correctly handle negative SWI numbers in RISC OS 2.

Related SWIs

OS_SWINumberFromString (page 1-474)

Related vectors

None

OS_SWINumberFromString (SWI &39)

Convert a string to a SWI number if valid

On entry

R1 = pointer to name (which is terminated by a character ≤ 32)

On exit

R0 = SWI number
R1 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_SWINumberFromString converts a SWI name to a SWI number. An error is given if the SWI name is not recognized.

The conversion is as follows:

- A leading X is checked for and stripped. If present, &20000 is added to the number returned (ie bit 17 will be set).
- System names are checked for. Note that the conversion of SWIs is not quite bidirectional: the name OS_WriteI+" " can be produced, but only OS_WriteI is recognized.
- Modules are scanned. If the module prefix matches the one given, and the suffix to the name is a number, then that number is added to the module's SWI 'chunk' base, and the sum returned. For example, Wimp_&23 returns &400E3, as the Wimp's chunk number is &400C0.

- If the suffix is a name, and this can be matched by the module, the appropriate number is returned. For example, Wimp_Poll returns &400C7.

See the chapter entitled *Modules* on page 1-201 for more information on how modules provide the conversion.

Note that SWI names are case sensitive, so you must spell them exactly as returned by OS_SWINumberToString.

Related SWIs

OS_SWINumberToString (page 1-472)

Related vectors

None

OS_SubstituteArgs (SWI &43)

Substitute command line arguments

On entry

R0 = pointer to argument list, and flag in top bit
R1 = pointer to buffer for result string
R2 = length of buffer
R3 = pointer to template string
R4 = length of template string

On exit

R0, R1 preserved
R2 = number of characters in result string (including the terminator)
R3, R4 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 performs the hard work involved in substituting a list of arguments into a ‘template’ string. Its main use is in the processing of command Alias$… variables by the system. As it is also useful in other situations, it has been made available to users. For example, FileSwitch uses it in the processing of Alias$@LoadType_TTT variables.

The argument list is a string consisting of space-separated items which will be substituted into the template string. Spaces within double quotation marks are not counted as argument separators. Typically, the argument string will just be the tail of a * Command. It is control-character terminated.

The result of substituting the arguments into the template string is placed in the buffer. The length of the buffer is given so that the call can check for buffer overflow.

The template string is copied into the result buffer character for character. However, when a '%' appears in the template string (even within quotation marks), it marks where an argument should be placed into the output buffer. The '%' is followed by a single digit from 0 to 9. %0 stands for the first argument in the argument list, and so on. %*n* means all of the arguments from the *n*th one onwards. %% means a single '%'. Anything else following the '%' is not treated specially, ie both the '%' and the character are copied over.

The template string does not have a terminator; instead its length is given. At the end of the substitution, any arguments after the highest one mentioned in the template string are appended to the result string. This can be stopped by setting the top bit of R0 on entry.

If a non-existent argument is specified in the template string, then a null string is substituted; no error is given.

Related SWIs

None

Related vectors

None

OS_ReadArgs (SWI &49)

Given a keyword definition, scan a command string

On entry

R0 = pointer to keyword definition
R1 = pointer to input string
R2 = pointer to output buffer
R3 = size of output buffer

On exit

R0 - R2 preserved
R3 = bytes left in output buffer

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI processes a command string using a keyword definition for syntax. The results are written out to the output buffer using a specialised format for this command.

Keyword definition

The keyword definition defines the parameters that can be in the command string. It is composed of a sequence of keywords, separated by commas. Each of these is made up of one or two names, followed by a sequence of qualifiers. The syntax of a keyword is:

[*keyword_name*[=*alias_name*]][/*qualifier*...]

The *keyword_name* is what you want users to identify the parameter with. This can be any string composed of alphanumerics and the '_' character. The alias name is an optional alternative name for the same keyword. You can have a keyword with no name. See the command string description below for details of how to set it.

The qualifier describes what kind of a parameter it is. There can be as many qualifiers as you like with one parameter, but some are mutually exclusive. The qualifiers can be any one of the following characters in upper or lower case:

/A	keyword must always be given a value
/K	keyword must always precede its value
/S	the option is a switch ie presence only is reported
/E	OS_EvaluateExpression will be called to transform the value. This can return a number or a string. Note that numeric evaluations only can be performed. Note that in RISC OS 2 a 'Buffer full' error is generated if this argument evaluates to a string
/G	OS_GSTrans will be called to transform the value

Command string

The command string contains a sequence of commands using the syntax defined by the keyword definition. A command string is made of definitions of the following syntax:

[–*keyword_name*] *value*

If the keyword name is used, then the value will be attached to the named keyword. These can appear in any arbitrary order in the command string. The name after the '–' can be the full name of the keyword or its alias, or the first letter of either. For example, if the keyword definition contains "name=title", then all of the following are valid in the command string:

"–name fred", "–title fred", "–n fred", "–t fred"

Note that if more than one keyword has the same first letter, then the single letter form will be used by the first occurrence of a given letter in the keyword definition.

Also note that case is ignored, so "–FILE" and "–file" are identical.

If a definition has no *–keyword_name* preceding it, then the first unused keyword that is not a switch in the definition string will be given that value. This is how nameless keywords are set. For example, if the definition string is "infile,/a,outfile" and the command string is "–infile one –outfile two three", then the first and nameless keyword will be set to three, because it was the first undefined keyword in the definition.

Keywords are marked by a preceding ‘–’ character, but this does not disallow these characters from appearing in values anywhere but at the start. For example, if the keyword definition is "formula/e", then "–formula 6–3" will set it to the value of 3. If the command is "–formula –3+6", then this will cause an error. Furthermore, whilst some evaluated expressions can be done without spaces (1+2 for example), there are many that cannot.

A workround for these problems is to evaluate your expression in quotes, which allow leading minuses, or spaces – as in this example:

```
"&3F AND &17"
```

With GSTrans’d strings, if you want to put a quoted string inside quotes then you must use double quotes, as follows:

```
"This is ""IT"""
```

Output buffer

The output buffer contains the results for all of the possible keywords. For *n* keywords in the keyword definition, the first *n* words of the output buffer contain values giving the results of parsing the command line. If the keyword was a switch (with /S qualifier), then a non-zero value indicates that the switch was used. For all other kinds of result, the value is a pointer; the actual results are appended sequentially to the output buffer. A pointer of zero indicates that the parameter was not present.

The following example uses a keyword definition of "ax,bx,on/s,cx" and a command string of "one two three –on". The output buffer looks like this:

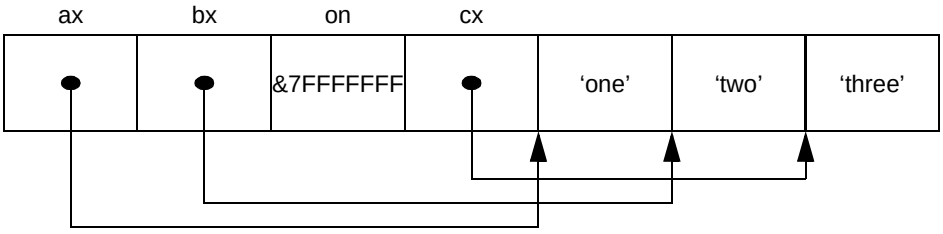


Figure 18.1 Example output buffer

The results of GSTrans’d strings and evaluated expressions are stored differently. In a GSTrans’d string, the result pointer points to a block of the following format:

length	two byte length
string	*length* bytes of string

In an evaluated expression, the pointer points to a block like the following:

type	one byte result type (which at present can only be zero for an integer)
value	four byte integer

For an example showing /e and /g switches, if the keyword definition was "formula/e,time/g" and the command string was "–f 6+6–1 –t ""Time is <Sys$Time>""", then the result looks like this:

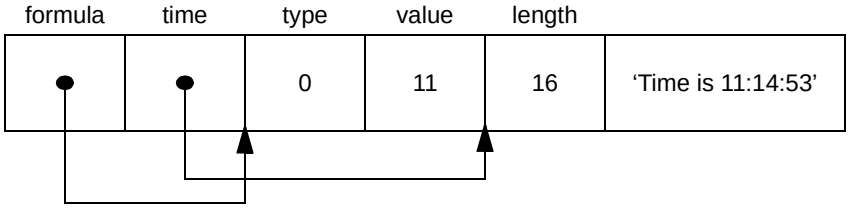


Figure 18.2 Example output buffer

Examples

Keyword definition:

```
number=times/e,file/k/a,expandtabs/s
```

can be matched by any of:

```
–n 10 –file jeff
–times 1+7 –file jeff –expandtabs
–file thingy –e
```

but not by either of:

```
thingy –number 4
–number 20 –times 4 –file jeff
```

Related SWIs

None

Related vectors

None

OS_Convert*NameNumber* (SWIs &D0 - E8)

These calls convert a number into a string

On entry

R0 = value to be converted
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWIs are re-entrant

Use

This range of SWIs use a common form and can convert a number into a string in a variety of ways.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

The *Name* part of the SWI name can be any of the following groups:

Hex

Convert to a hexadecimal string

The *Number* is the number of ASCII digits in the output string, either 1, 2, 4, 6 or 8. Only enough significant bits to perform the conversion are used. The string does not include an ampersand ('&') but is padded with leading zeros, so is of fixed length. The SWIs in this group are:

SWI name	SWI number	Output for… zero	largest value
OS_ConvertHex1	&D0	'0'	'F'
OS_ConvertHex2	&D1	'00'	'FF'
OS_ConvertHex4	&D2	'0000'	'FFFF'
OS_ConvertHex6	&D3	'000000'	'FFFFFF'
OS_ConvertHex8	&D4	'00000000'	'FFFFFFFF'

Cardinal

Convert to an unsigned decimal number

The *Number* is the number of bytes to be used from the input value. The string is not padded with zeros, so is of variable length. The SWIs in this group are:

SWI name	SWI number	Output for… zero	largest value
OS_ConvertCardinal1	&D5	'0'	'255'
OS_ConvertCardinal2	&D6	'0'	'65535'
OS_ConvertCardinal3	&D7	'0'	'16777215'
OS_ConvertCardinal4	&D8	'0'	'4294967295'

Integer

Convert to a signed decimal number

The *Number* is the number of bytes to be used from the input value. If the most significant bit is set (of the *number* bytes used), the number is taken to be negative, and a leading '–' is produced. The string is not padded with zeros, so is of variable length. The SWIs in this group are:

SWI name	SWI number	Output for… largest –ve	largest +ve value
OS_ConvertInteger1	&D9	'–128'	'127'
OS_ConvertInteger2	&DA	'–32768'	'32767'
OS_ConvertInteger3	&DB	'–8388608'	'8388607'
OS_ConvertInteger4	&DC	'–2147483648'	'2147483647'

Binary

Convert to a binary number

The *Number* is the number of bytes to be used from the input value. The string is padded with leading zeros, so is of fixed length (*number* × 8). The SWIs used in this group are:

SWI name	SWI number	Output for largest value
OS_ConvertBinary1	&DD	'11111111'
OS_ConvertBinary2	&DE	'1111111111111111'
OS_ConvertBinary3	&DF	'111111111111111111111111'
OS_ConvertBinary4	&E0	'11111111111111111111111111111111'

SpacedCardinal

Convert to an unsigned decimal number, with spaces every three digits

The *Number* is the number of bytes to be used from the input value. The string is not padded with zeros, so is of variable length. In addition, every three digits from the right, a space is inserted. The SWIs used in this group are:

SWI name	SWI number	Output for… zero	largest value
OS_ConvertSpacedCardinal1	&E1	'0'	'255'
OS_ConvertSpacedCardinal2	&E2	'0'	'65 535'
OS_ConvertSpacedCardinal3	&E3	'0'	'16 777 215'
OS_ConvertSpacedCardinal4	&E4	'0'	'4 294 967 295'

SpacedInteger

Convert to a signed decimal number, with spaces every three digits

The *Number* is the number of bytes to be used from the input value. If the most significant bit is set (of the *number* bytes used), the number is taken to be negative, and a leading '–' is produced. The string is not padded with zeros, so is of variable length. In addition, every three digits from the right, a space is inserted. The SWIs in this group are:

SWI name	SWI no.	Output for… largest –ve	largest +ve val.
OS_ConvertSpacedInteger1	&D9	'–128'	'127'
OS_ConvertSpacedInteger2	&DA	'–32 768'	'32 767'
OS_ConvertSpacedInteger3	&DB	'–8 388 608'	'8 388 607'
OS_ConvertSpacedInteger4	&DC	'–2 147 483 648'	'2 147 483 647'

Related SWIs

OS_BinaryToDecimal (page 1-468)

Related vectors

None

OS_ConvertFixedNetStation (SWI &E9)

Convert from an Econet station/network number pair to a string

On entry

R0 = pointer to two word block (value to be converted)
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null zero in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

R0 points to two words in memory. The first word contains the station number and the second word contains the network number.

This call always converts into a form *nnn.sss*, where *nnn* is the network number and *sss* the station number. If the network number is zero, the first four characters are spaces; if it is non-zero, leading zeros are converted to spaces. If the network number was zero, leading zeros in the station number are converted to spaces; otherwise they are left as zeros.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertNetStation (page 1-488)

Related vectors

None

OS_ConvertNetStation (SWI &EA)

Convert from an Econet station/network number pair to a string

On entry

R0 = pointer to two word block (value to be converted)
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

R0 points to two words in memory. The first word contains the station number and the second word contains the network number.

This call performs the same conversion as OS_ConvertFixedNetStation, but suppresses zeros and spaces wherever possible, to yield the shortest possible string.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertFixedNetStation (page 1-486)

Related vectors

None

OS_ConvertFixedFileSize (SWI &EB)

Convert an integer into a filesize string of a fixed length

On entry

R0 = filesize in bytes
R1 = pointer to buffer
R2 = length of buffer in bytes

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

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 SWI will convert an integer into a filesize string of a fixed length. The format of the string is:

4_digit_number `M|K|`*space*`bytes`*null*

The *4_digit_number* at the start is padded with spaces if it is less than four digits in length; *space* and *null* are the ASCII characters 32 and 0 respectively.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertFileSize (page 1-492)

Related vectors

None

OS_ConvertFileSize (SWI &EC)

Convert an integer into a filesize string

On entry

R0 = filesize in bytes
R1 = pointer to buffer
R2 = length of buffer in bytes

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

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 SWI will convert an integer into a filesize string. The format of the string is:

number `[M|K]bytes`*null*

The *number* at the start is up to four digits in length; *null* is the ASCII character 0. For small sizes the optional ‘M’ or ‘K’ is omitted.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertFixedFileSize (page 1-490)

Related vectors

None

*Commands

*Echo

Displays a string on the screen (after translating it using OS_GSTrans)

Syntax

`*Echo` *`string`*

Parameters

`string`	string to display

Use

*Echo takes the string following it, translates it using OS_GSTrans, and then displays it on the screen.

The main use for *Echo is in command scripts, where the command provides a useful way of checking the progress of a script, especially when debugging a faulty script, or when monitoring the progress of a series of operations.

Example

```
*Echo |GError!|M
```

Related commands

None

Related SWIs

None

Related vectors

None

*Eval

Evaluates an integer, logical, bit or string expression

Syntax

`*Eval` *`expression`*

Parameters

`expression`	any combination of the operators listed earlier

Use

*Eval evaluates an integer, logical, bit or string expression, carrying out type conversions where necessary, in a similar way to the BASIC EVAL command. It will not handle floating point numbers. You can use *Eval to do simple arithmetic (although the desktop Calculator is easier to use for four-function arithmetic), or to evaluate more complex expressions. Programmers may find the command useful for doing ‘off-line’ calculations (checking on remaining space, for example).

See the section entitled *Evaluation operators* on page 1-456 for a description of the operators that you can use. Note that monadic plus/minus operators are not correctly handled in RISC OS 2 (eg *Eval 50*–3 gives a ‘Missing operand’ error).

Example

```
*Eval 127 * 23 >> 2
Result is an integer, value : 730
```

Related commands

*If, *SetEval

Related SWIs

OS_EvaluateExpression (page 1-470)

Related vectors

None

*If

Conditionally executes a * Command, depending on the value of an expression

Syntax

```
*If expression Then command [Else command]
```

Parameters

expression	an integer expression
command	any valid * Command

Use

*If conditionally executes a * Command, depending on the value of an expression. The expression can be any integer expression, including (if necessary) variable names enclosed in angled brackets.

The expression is evaluated by the operating system's expression evaluation. If the If-expression evaluates to a non-zero value, the Then-clause is executed. If the If-expression evaluates to zero, and there is an Else-clause, the Else-clause is evaluated.

If you wish to compare a variable to a string both must be enclosed in double quotes to ensure a string comparison is performed; see the first example.

See the section entitled *Evaluation operators* on page 1-456 for a description of the operators that you can use.

Example

```
*If "<name>" = "Michael" Then Echo Hi Mike! Else Echo Go away <name>!

If <Sys$Year>=1992 Then Run Calendar
```

Related commands

*Eval

Related SWIs

None

Related vectors

None

19 Extension ROMs

Extension ROMs are ROMs fitted in addition to the main ROM set, which provide software modules which are automatically loaded by RISC OS on a reset. Note that **RISC OS 2 does not support extension ROMs**.

Extension ROMs are provided so that Acorn can add extra modules to RISC OS, or provide replacement modules for those already in RISC OS. **You must not use them**. It is the Expansion Card Manager's responsibility to recognise extension ROMs. For it to do so, extension ROMs need to have headers, which are detailed in the chapter entitled *Expansion Cards and Extension ROMs* on page 4-117. That chapter also gives details of the software that RISC OS provides to manage and communicate with extension ROMs (and, of course, expansion cards). Expansion cards and extension ROMs are covered together because both use substantially the same layout of code and data, and the same SWIs.

It is the kernel's responsibility to load any relocatable modules from any extension ROMs – once the Expansion Card Manager has recognised them. For your information, the chapter entitled *Modules* on page 1-201 gives details of how the modules are loaded on a reset.

Part 3 – Kernel input/output

20 Character Output

Introduction

The Character Output system can send characters to the computer's 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 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 set up 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 by default 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* on page 2-393 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* on page 3-565 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, although not all machines will support different transmit and receive rates.

Calls that are specific to the serial port, whether they refer to input or output (eg those to set the baud rate, or to explicitly send/receive a character from/to the serial port), are gathered together in the chapter entitled *Serial device* on page 2-445.

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* on page 1-547.

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 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 still one way of 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.

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 Drivers* on page 1-547.

The flow of control of the filtering – which controls which characters sent to the VDU stream also get sent to the printer – is summarised by the diagram below:

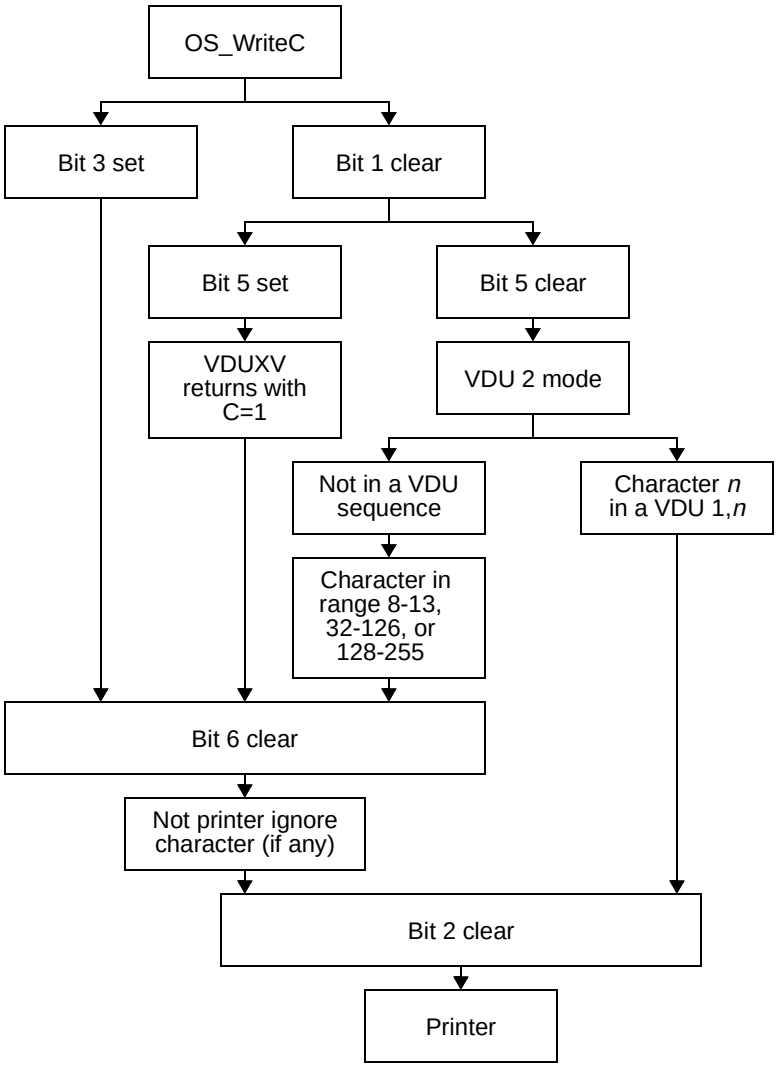


Figure 20.1 Flow of control of filtering in the printer stream

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 can be used to set the printer ignore character from the CLI. *Configure Ignore 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 with no other parameters will set the NoIgnore state permanently in CMOS RAM.

Serial device

The serial device is provided as a DeviceFS (*Device Filing System*) device. For full details, see the chapter entitled *DeviceFS* on page 2-429, and the chapter entitled *Serial device* on page 2-445. The latter chapter also contains all calls that are specific to the serial port, whether they refer to input or output – such as those to set the baud rate, or to explicitly send/receive a character to/from the serial port.

(Under RISC OS 2, the serial device was provided by the SystemDevices module. See the section entitled *The RISC OS 2 serial device* on page 2-497.)

OS_Byte calls in this chapter

OS_Bytes 3 and 5 can be used to select the serial port 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. (However, there are preferred calls for sending a byte to the serial port; see the chapter entitled *Serial device* on page 2-445.)

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.

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 or *SpoolOn 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.

Character	Meaning
0 - 31	VDU commands (graphics and control)
32 - 126	ASCII characters
127	Delete
128 - 159	Acorn defined characters, and user definable characters
160 - 255	ISO international characters

Note that if defining characters in the range 128 - 159 under the Desktop, you should always first read the current definition of the character 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* on page 1-63 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 on page 1-744.

SWI Calls

OS_WriteC (SWI &00)

Writes 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 (see page 1-66), 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 default ROM routine, or if a spool file is active, or if the printer is active etc.

Related SWIs

OS_WriteS (page 1-517), OS_Write0 (page 1-518), OS_NewLine (page 1-519), OS_PrettyPrint (page 1-536), OS_WriteN (page 1-540), OS_WriteI (page 1-541), OS_Byte 3 (page 1-520)

Related vectors

WrchV

OS_WriteS (SWI &01)

Writes 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 byte.

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 (page 1-515)

Related vectors

WrchV

OS_Write0 (SWI &02)

Writes an indirect string to all of the active output streams

On entry

R0 = pointer to null-terminated string to write

On exit

R0 = pointer 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. The string is terminated by a null byte.

Related SWIs

OS_WriteC (page 1-515)

Related vectors

WrchV

OS_NewLine (SWI &03)

Writes 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 writes a line feed followed by a carriage return to all of the active output streams. It uses two calls to OS_WriteI to do so, which in turn call OS_WriteC.

Related SWIs

OS_WriteC (page 1-515), OS_WriteI (page 1-541)

Related vectors

WrchV

OS_Byte 3 (SWI &06)

Selects the output streams that are active

On entry

R0 = 3 (reason code)
R1 = bit mask for output streams

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 which output stream(s) are active, and will hence receive all subsequent output. A bit mask in R1 determines this:

Bit	Effect if set
0	Enables serial driver
1	Disables VDU drivers
2	Disables VDU printer stream
3	Enables printer (independently of the VDU)
4	Disables spooled output
5	Calls VDUXV instead of VDU drivers (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 drivers, the VDU printer stream and the spool stream are enabled, and other streams disabled

Details of how bits 1, 2, 3 and 6 interact is described in the section entitled *Technical Details* on page 1-508 onwards.

Related SWIs

OS_Byte 236 (page 1-530)

Related vectors

ByteV, VDUXV, WrchV

OS_Byte 5 (SWI &06)

Sets which PrinterType$… variable holds the printer output path

On entry

R0 = 5 (reason code)
R1 = number *n* of PrinterType$*n* variable to use (0 - 255)

On exit

R0 = preserved
R1 = previous printer driver type
R2 = corrupted

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets which PrinterType$… variable holds the path to use for subsequent printer output.

For R1 = *n* the path held in the variable PrinterType$*n* is used. The default values of these variables route printed output as follows:

n	**Printer**
0	Null (no output)
1	Parallel port
2	Serial port
3	Path in system variable PrinterType$3 (reserved for user printer driver)
4	Network printer (handled through NetPrint)
5-255	Paths in system variables PrinterType$5 to PrinterType$255

Under RISC OS 2 values of 0, 1 or 2 explicitly set output as given above, rather than by consulting the variables PrinterType$0, $1 and $2.

The default variable to use is set by *Configure Print; for this purpose, *n* is restricted to the range 0 - 7.

Note that if the current PrinterType$… variable is set to the serial device's path, 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.

The new PrinterType$… variable comes into effect only when all the current contents of the printer buffer have been sent to the path held in the previously selected variable. 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. (You may force this to happen by acknowledging an escape condition from the foreground, provided that the escape side effects are enabled.)

Related SWIs

OS_Byte 8 (page 2-453), OS_Byte 245 (page 1-532)

Related vectors

ByteV

OS_Byte 6 (SWI &06)

Sets the printer ignore character

On entry

R0 = 6 (reason code)
R1 = ASCII code 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

This call sets the printer ignore character to the specified ASCII code. This character is filtered out when printing is enabled via the VDU printer stream or OS_Byte 5(page 1-522).

The default value of the printer ignore character is set by *Configure Ignore. You may temporarily change it using this OS_Byte, or *Ignore. The latter has the advantage that it also allows a *NoIgnore* state to be set.

Related SWIs

OS_Byte 5 (page 1-522), OS_Byte 182 (page 1-526), OS_Byte 246 (page 1-534)

Related vectors

ByteV

OS_Byte 182 (SWI &06)

Reads/writes the printer *NoIgnore* state

On entry

R0 = 182 (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 Š&80 (ie has bit 7 set), then the printer ignore character is not used. If bit 7 is clear, then the current printer ignore character is filtered out.

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

Related SWIs

OS_Byte 6 (page 1-524), OS_Byte 246 (page 1-534)

Related vectors

ByteV

OS_Byte 199 (SWI &06)

Reads/writes the spool file handle

On entry

R0 = 199 (reason code)
R1 = 0 to read or new handle (as returned by OS_Find) to write
R2 = 255 to read or 0 to write

On exit

R0 = preserved
R1 = handle 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 reads/writes the spool file handle, which sets the destination of spooled data. The handle must have been correctly returned from a previous call to OS_Find (page 2-75). If the file handle is zero, or if spooling is disabled by OS_Byte 3, then no spooled data is sent.

Related SWIs

OS_Byte 3 (page 1-520), OS_Find (page 2-75)

Related vectors

ByteV

OS_Byte 236 (SWI &06)

Read/write character destination status

On entry

R0 = 236 (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, page 1-929)

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 stream's value. This can also be written by OS_Byte 3. See OS_Byte 3 for a list of the bit values.

Related SWIs

OS_Byte 3 (page 1-520)

Related vectors

ByteV

OS_Byte 245 (SWI &06)

Reads which PrinterType$… variable holds the printer output path

On entry

R0 = 245 (reason code)
R1 = 0
R2 = 255

On exit

R0 = preserved
R1 = value before being overwritten
R2 = value of printer ignore character (see OS_Byte 246, page 1-534)

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 which PrinterType$… variable holds the path to use for subsequent printer output

For R1 = *n* the path held in the variable PrinterType$*n* is used. The default values of these variables route printed output as follows:

n	**Printer**
0	Null (no output)
1	Parallel port
2	Serial port

3	Path in system variable PrinterType$3 (reserved for user printer driver)
4	Network printer (handled through NetPrint)
5-255	Paths in system variables PrinterType$5 to PrinterType$255

Under RISC OS 2 values of 0, 1 or 2 explicitly set output as given above, rather than by consulting the variables PrinterType$0, $1 and $2.

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 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 (page 1-522)

Related vectors

ByteV

OS_Byte 246 (SWI &06)

Read/write printer ignore character

On entry

R0 = 246 (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.

Related SWIs

OS_Byte 6 (page 1-524), OS_Byte 182 (page 1-526)

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 (page 1-518), 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, which 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 content 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 (page 1-515)

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 sends 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, page 1-524) 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

OS_WriteC (page 1-515), OS_WriteS (page 1-517), OS_WriteO (page 1-518)

Related vectors

WrchV

OS_WriteI (SWIs &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 (page 1-515)

Related vectors

WrchV

*Commands

*Configure Ignore

Sets the configured printer ignore character

Syntax

`*Configure Ignore [`*`ASCII_code`*`]`

Parameters

`ASCII_code`	ASCII code, from 0 to 255

Use

*Configure Ignore sets the configured printer ignore character to the specified ASCII code. This character is filtered out when printing is enabled via the VDU printer stream or OS_Byte 5. With no parameter, the NoIgnore state is configured, so all characters will be printed.

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 ASCII code so all characters are sent to the printer. *Configure Ignore 0 will not ensure all characters are printed; it will set the configured printer ignore character to ASCII 0 (the null character).

The change takes effect on the next hard reset.

Example

`*Configure Ignore 10`	*Do not print ASCII character 10*
`*Configure Ignore`	*Print all characters*

Related commands

*Ignore

Related SWIs

OS_Byte 6 (page 1-524), OS_Byte 5 (page 1-522), OS_Byte 182 (page 1-526), OS_Byte 246 (page 1-534)

Related vectors

None

*Configure Print

Sets the configured default destination for printed output

Syntax

```
*Configure Print n
```

Parameters

n 0 to 7

Use

*Configure Print sets the configured default destination for printed output. For a value *n* the path held in the variable PrinterType$*n* is used. The default values of these variables route printed output as follows:

n	Printer
0	Null (no output)
1	Parallel port
2	Serial port
3	Path in system variable PrinterType$3 (reserved for user printer driver)
4	Network printer (handled through NetPrint)
5-7	Paths in system variables PrinterType$5, 6 or 7

Under RISC OS 2 values of 0, 1 or 2 explicitly set output as given above, rather than by consulting the variables PrinterType$0, $1 and $2.

The change takes effect on the next hard reset.

Example

```
*Configure Print 1
```
select the parallel printer port

Related commands

None

Related SWIs

OS_Byte 5 (page 1-522)

Related vectors

None

*Ignore

Sets the printer ignore character

Syntax

`*Ignore [`*`ASCII_code`*`]`

Parameters

`ASCII_code`	ASCII code, from 0 to 255

Use

*Ignore sets the printer ignore character to the specified ASCII code. This character is filtered out when printing is enabled via the VDU printer stream or OS_Byte 5. With no parameter, the NoIgnore state is set, so all characters are printed.

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 ASCII code so all characters are sent to the printer. *Ignore 0 will not ensure all characters are printed; it will set the printer ignore character to ASCII 0 (the null character).

OS_Byte 6 performs the same action as this command; OS_Byte 246 also reads and writes the printer ignore character. OS_Byte 182 controls the *NoIgnore* state.

Example

`*Ignore 10`	*Do not print ASCII character 10*
`*Ignore`	*Print all characters*

Related commands

None

Related SWIs

OS_Byte 5 (page 1-522), OS_Byte 6 (page 1-524), OS_Byte 182 (page 1-526), OS_Byte 246 (page 1-534)

Related vectors

None

21 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* on page 1-503 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* on page 1-594, 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* on page 1-863 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.

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 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 (page 1-515) 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 entitled *Sprites* on page 1-773 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 method 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 complementary 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.

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 eight 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, an 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 (page 1-594).

*Configure Mode (page 1-757) 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 (page 1-742) 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 (page 1-759) is used to tell RISC OS to auto-detect the type of monitor connected from its lead, or – should it be unable to do so from hardware – to tell it explicitly what kind of monitor is attached. This command allows the system to subsequently disallow any modes that are not compatible with the attached monitor.

*Configure Sync (page 1-766) will set up the vertical sync output of the video connector to be auto-detected, or to be vertical or composite sync. Different monitors may require either of these, though most use composite sync.

*Configure TV (page 1-767), *TV (page 1-769) and OS_Byte 144 (page 1-671) can all adjust the position of the video output up or down by several lines, and switch interlace on and off. VDU 23,0 (page 1-600) 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 (page 1-660) selects which bank of memory to send VDU output to. OS_Byte 113 (page 1-662) 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 (page 1-696) reads the current OS_Byte 112 setting, and OS_Byte 251 (page 1-698) reads the current OS_Byte 113 setting.

In order to use multiple banks, you will probably have to use *Configure ScreenSize (page 1-764) to set the amount of memory to reserve for all the banks.

*Shadow (page 1-768) 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 (page 1-664) 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 (page 1-724) 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 (page 1-585) can be used to change the text colour. VDU 23,17,5| will exchange the text foreground and background colours.

VDU 18 (page 1-586) 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 (page 1-588) 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 pointer colours. OS_Word 12 can also be used to write the palette.

VDU 20 (page 1-592) 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 (page 1-728) 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 (page 1-649). They set the duration in video frames. VDU 23,9 and VDU 23,10 (page 1-608) have the same effect as these calls. The duration settings can be read by OS_Bytes 194 and 195 (page 1-680).

OS_Byte 193 (page 1-678) 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 on 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 trade-off 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 section entitled *Application Notes* on page 1-770.

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 is aligned with the bottom left hand corner of the screen. Using OS_SetECFOrigin, you can instead align an ECF pattern to any point on the screen, or to an 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 (page 1-684) will select the sound channel used
- OS_Byte 212 (page 1-686) will adjust the volume
- OS_Byte 213 (page 1-688) will adjust the frequency
- OS_Byte 214 (page 1-690) will adjust the duration

*Configure Quiet will select a quiet volume, while *Configure Loud will select a loud volume.

Cursors

VDU 5 (page 1-573) 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 (page 1-572). 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 (page 1-635) – send the text cursor to its home position, which is usually the top left corner of the current window.
- VDU 31 (page 1-636) – set the text cursor to any position on the screen.
- VDU 8 (page 1-576) – backspace
- VDU 9 (page 1-577) – horizontal tab
- VDU 10 (page 1-578) – linefeed (ie move down)
- VDU 11 (page 1-579) – vertical tab (ie move up one line)
- VDU 13 (page 1-581) – move back to the start of the line
- VDU 127 (page 1-637) – delete (ie backspace, print a space then backspace again).

The position of the text cursor can be read with OS_Byte 134 (page 1-668). If cursor editing is in progress, then OS_Byte 165 (page 1-677) 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 (page 1-739) will remove the input and output cursors and store their state internally. A subsequent call to OS_RestoreCursors (page 1-741) 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 (page 1-708) will return the current and previous graphics cursor positions. Using OS_ReadVduVariables (page 1-730), 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 (page 1-726) 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 (page 1-716) 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 (page 1-718) 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 (page 1-714) or permanently set by *Configure MouseStep (page 1-761).

The pointer that appears on the screen can be defined in four shapes. OS_Word 21,0 (page 1-710) can define the shape and colour of each of these. OS_Byte 106 (page 1-658) 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 (page 1-712). This would usually be set to the graphics window.

The pointer's position on the screen can be set with OS_Word 21,5 (page 1-720) and read with OS_Word 21,6 (page 1-722).

Getting information

There are many ways of extracting information about the state and configuration of the VDU system.

OS_Byte 217 (page 1-692) will read the number of lines since the display was last stopped scrolling if it was in paged mode.

OS_Byte 218 (page 1-694) returns how many bytes are in the VDU queue. This is used when a multiple byte VDU command is being collected.

OS_Byte 163 (page 1-675) 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 (page 1-396) 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 (page 1-666) 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 (page 1-730) 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 (page 1-736) 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 (page 1-670) will read the ASCII value of the character at the text cursor position and also reads the current screen mode.

OS_ReadPoint (page 1-734) 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 (page 1-847) will copy the screen contents into a file where it can subsequently be edited with Paint or reloaded to the screen with *ScreenLoad (page 1-846).

Writing to the screen

Output to the screen can be disabled by VDU 21 (page 1-593). It can be restored by VDU 6 (page 1-574).

VDU 26 (page 1-631) 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 (page 1-633) defines the text window. VDU 12 (page 1-580) 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 (page 1-606) will clear a block within the text window.

Paged 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 (page 1-582). By default, paged mode is off.

*Configure Scroll (page 1-765) and NoScroll (page 1-762) 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 (page 1-614) and allows a full screen of text to be simply printed.

VDU 23,7 (page 1-604) 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 (page 1-620). 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 (page 1-625) 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 (page 1-654) will reset character definitions 32 - 127 to their default. OS_Byte 25 (page 1-656) will reset a given group of them. OS_Word 10 (page 1-702) can read the definition of any character from the current system font.

Printer

VDU 1 (page 1-569) will send the following character to the printer stream. VDU 2 (page 1-570) 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 (page 1-571).

Graphics

VDU 24 (page 1-627) will define the position of the graphics window. VDU 16 (page 1-584) will clear it to the current graphics background colour.

VDU 25 (page 1-628) is the main graphics plot command. OS_Plot (page 1-744) 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 (page 1-603) is passed 8 bytes that define a pattern up to 64 bits in length to be repeated. OS_Byte 163 (page 1-675) 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 (page 1-702) can read the current definition.

VDU 29 (page 1-634) sets the graphics origin. This is the point on the screen that becomes the 0,0 point for all subsequent graphics operations.

OS_ChangedBox (page 1-752) 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 (page 1-846) complements *ScreenSave, discussed earlier and load a file into the screen memory.

Vsync

OS_Byte 19 (page 1-652) 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.

Screen memory and hardware scrolling

This is described fully in the section entitled *Screen memory* on page 1-345.

VDU Calls

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 linefeed 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, as described on page 1-522. Only characters in the following ranges are sent to the printer: 32 - 126, 128 - 255 (ie the printable characters), 8 - 13 (backspace, horizontal tab, linefeed, 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. Again, you can use VDU 23,16 to enable or disable scrolling. Cursor editing is allowed in this mode.

You can determine whether the cursors are split or joined using OS_Byte 117 (page 1-666).

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 (page 1-666).

VDU 7

Bell

Syntax

```
VDU 7
```

Parameters

—

Use

VDU 7 generates the current bell sound. The initial default is specified by *Configure Loud/Quiet and *Configure SoundDefault; you may subsequently alter it using OS_Bytes 211 - 214.

VDU 8

Backspace

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 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

Linefeed

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 linefeed.

VDU 14

Paged 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 (page 1-666) may be used to determine whether paged mode is enabled. See also OS_Byte 217 (page 1-692).

VDU 15

Paged 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 bit 3 Bit 4 Blue bit 2 Bit 3 Green bit 3 Bit 2 Green bit 2 Bit 1 Red bit 3 Bit 0 Red bit 2

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 (page 1-730).

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 = physical colour specified by mode parameter (see below); red, green and blue are ignored, and should be zero
mode = 16	both flash palettes for logical colour = red units red, green units green, blue units blue
mode = 17	first flash palette for logical colour = red units red, green units green, blue units blue
mode = 18	second flash palette for logical colour = red units red, green units green, blue units blue
mode = 24	border colour = red units red, green units green, blue units blue; logical colour is not used, and should be zero
mode = 25	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 physical colours specified by the mode parameter when it is in the range 0 - 15 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:

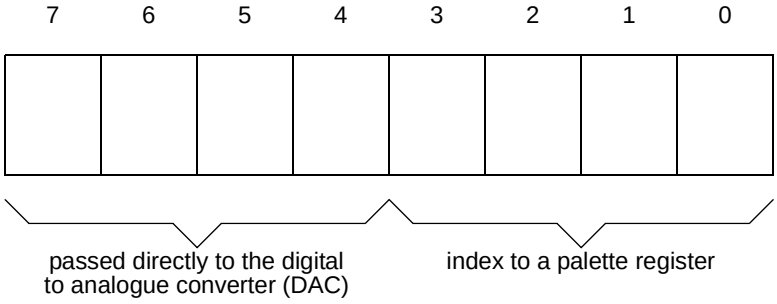


Figure 21.1 Treatment of logical colours in eight bit per pixel modes

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

By default, the palette registers 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 cannot be used to produce extreme 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 (page 1-706), and read the current palette using OS_Word 11 (page 1-704) and OS_ReadPalette (page 1-728).

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 (page 1-666) 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 bottom seven bits of the mode parameter are used to select the mode. The modes available in RISC OS depend on the configured monitor type (see *Configure MonitorType on page 1-759) and the model of computer. Below is a table of all modes provided by RISC OS, which shows:

- the mode number
- the text resolution in columns × rows
- the graphics resolution in pixels, which corresponds to the clarity of the mode's display
- the resolution in OS units, which corresponds to the area of workspace shown by the mode
- the number of logical colours available
- the memory used to display the screen (to the nearest 0.1Kbyte)
- the vertical refresh rate to the nearest Hz (invalid for monitor type 5), which indicates the degree of flickering that you may perceive
- the bandwidth used to display the screen (to the nearest 0.1Mbyte/second), which corresponds to the load the mode places on the computer
- the monitor types that support that mode:

Type	Monitor	
0	50Hz TV standard colour or monochrome monitor	
1	Multi-frequency monitor	
2	64Hz high-resolution monochrome monitor	
3	60Hz VGA-type monitor	
4	Super-VGA-type monitor	(not available in RISC OS 2)
5	LCD (liquid crystal display)	(not available in RISC OS 2)

- the notes on the following page that are relevant to the mode.

Mode	Text resolution	Pixel resolution	OS units resolution	Logical colours	Mem used	Refresh rate	Band-width	Monitor types	Notes
0	80 × 32	640 × 256	1280 × 1024	2	20K	50Hz	1M/s	0,1,3,4,5	¬
1	40 × 32	320 × 256	1280 × 1024	4	20K	50Hz	1M/s	0,1,3,4,5	¬
2	20 × 32	160 × 256	1280 × 1024	16	40K	50Hz	2M/s	0,1,3,4,5	¬
3	80 × 25	Text only	Text only	2	40K	50Hz	2M/s	0,1,3,4,5	¬ƒý
4	40 × 32	320 × 256	1280 × 1024	2	20K	50Hz	1M/s	0,1,3,4,5	¬
5	20 × 32	160 × 256	1280 × 1024	4	20K	50Hz	1M/s	0,1,3,4,5	¬
6	40 × 25	Text only	Text only	2	20K	50Hz	1M/s	0,1,3,4,5	¬ƒý
7	40 × 25	Teletext	Teletext	16	80K	50Hz	4M/s	0,1,3,4,5	¬ƒ
8	80 × 32	640 × 256	1280 × 1024	4	40K	50Hz	2M/s	0,1,3,4,5	¬
9	40 × 32	320 × 256	1280 × 1024	16	40K	50Hz	2M/s	0,1,3,4,5	¬
10	20 × 32	160 × 256	1280 × 1024	256	80K	50Hz	4M/s	0,1,3,4,5	¬
11	80 × 25	640 × 250	1280 × 1000	4	40K	50Hz	2M/s	0,1,3,4,5	¬«
12	80 × 32	640 × 256	1280 × 1024	16	80K	50Hz	4M/s	0,1,3,4,5	¬
13	40 × 32	320 × 256	1280 × 1024	256	80K	50Hz	4M/s	0,1,3,4,5	¬
14	80 × 25	640 × 250	1280 × 1000	16	80K	50Hz	3.9M/s	0,1,3,4,5	¬«
15	80 × 32	640 × 256	1280 × 1024	256	160K	50Hz	8M/s	0,1,3,4,5	¬
16	132 × 32	1056 × 256	2112 × 1024	16	132K	50Hz	6.6M/s	0,1	Ý
17	132 × 25	1056 × 250	2112 × 1000	16	132K	50Hz	6.5M/s	0,1	Ý«
18	80 × 64	640 × 512	1280 × 1024	2	40K	50Hz	2M/s	1	
19	80 × 64	640 × 512	1280 × 1024	4	80K	50Hz	4M/s	1	
20	80 × 64	640 × 512	1280 × 1024	16	160K	50Hz	8M/s	1	
21	80 × 64	640 × 512	1280 × 1024	256	320K	50Hz	16M/s	1	
22	96 × 36	768 × 288	768 × 576	16	108K	50Hz	5.4M/s	0,1	¿¥
23	144 × 56	1152 × 896	2304 × 1792	2	126K	64Hz	8.1M/s	2	
24	132 × 32	1056 × 256	2112 × 1024	256	264K	50Hz	13.2M/s	0,1	Ý
25	80 × 60	640 × 480	1280 × 960	2	37.5K	60Hz	2.3M/s	1,3,4,5	
26	80 × 60	640 × 480	1280 × 960	4	75K	60Hz	4.5M/s	1,3,4,5	
27	80 × 60	640 × 480	1280 × 960	16	150K	60Hz	9M/s	1,3,4,5	
28	80 × 60	640 × 480	1280 × 960	256	300K	60Hz	18M/s	1,3,4,5	
29	100 × 75	800 × 600	1600 × 1200	2	58.6K	56Hz	3.3M/s	1,4	¿¡
30	100 × 75	800 × 600	1600 × 1200	4	117.2K	56Hz	6.6M/s	1,4	¿¡
31	100 × 75	800 × 600	1600 × 1200	16	234.4K	56Hz	13.2M/s	1,4	¿¡
33	96 × 36	768 × 288	1536 × 1152	2	27K	50Hz	1.4M/s	0,1	¿
34	96 × 36	768 × 288	1536 × 1152	4	54K	50Hz	2.7M/s	0,1	¿
35	96 × 36	768 × 288	1536 × 1152	16	108K	50Hz	5.4M/s	0,1	¿
36	96 × 36	768 × 288	1536 × 1152	256	216K	50Hz	10.8M/s	0,1	¿
37	112 × 44	896 × 352	1792 × 1408	2	38.5K	60Hz	2.3M/s	1	¿
38	112 × 44	896 × 352	1792 × 1408	4	77K	60Hz	4.6M/s	1	¿
39	112 × 44	896 × 352	1792 × 1408	16	154K	60Hz	9.2M/s	1	¿
40	112 × 44	896 × 352	1792 × 1408	256	308K	60Hz	18.5M/s	1	¿
41	80 × 44	640 × 352	1280 × 1408	2	27.5K	60Hz	1.7M/s	1,3,4,5	¿¬Ð
42	80 × 44	640 × 352	1280 × 1408	4	55K	60Hz	3.3M/s	1,3,4,5	¿¬Ð
43	80 × 44	640 × 352	1280 × 1408	16	110K	60Hz	6.6M/s	1,3,4,5	¿¬Ð
44	80 × 25	640 × 200	1280 × 800	2	15.7K	60Hz	0.9M/s	1,3,4,5	¿¬
45	80 × 25	640 × 200	1280 × 800	4	31.3K	60Hz	1.9M/s	1,3,4,5	¿¬
46	80 × 25	640 × 200	1280 × 800	16	62.5K	60Hz	3.8M/s	1,3,4,5	¿¬

Notes on display modes

1 These modes are not available in RISC OS 2.00, nor (except for mode 31) are they available in RISC OS 2.01.

2 These modes are not available on early models of RISC OS computers (ie the Archimedes 300, 400 and 400/1 series, and the A3000), because they are unable to clock VIDC at the necessary rate.

3 These modes are handled differently with a VGA or Super-VGA-type monitor. **If you are using such a monitor**:

- RISC OS 2.00 does not implement these modes.
- These modes are all displayed on a screen having 352 raster lines. Where a mode has fewer than 352 vertical pixels, it is centred on the screen with blank rasters at the top and bottom. Because of their appearance these modes are known as *letterbox modes*.
- The refresh rate is 70Hz.
- The bandwidths shown in the table for these modes are lower than these monitor types consume, because no allowance has been made for the blank rasters.
- Early models of RISC OS computers (ie the Archimedes 300, 400 and 400/1 series, and the A3000) scan these modes some 4.7% slow. Again this is because they are unable to clock VIDC at the necessary rate. Most VGA and Super-VGA-type monitors can still successfully lock onto this signal, but some may not. Furthermore, these models do not provide a *Sync Polarity* signal. This makes the effect of letterbox modes (see above) more severe.

4 Early models of RISC OS computers (ie the Archimedes 300, 400 and 400/1 series, and the A3000) also scan these modes some 4.7% slow with multi-frequency monitors. Again this is because they are unable to clock VIDC at the necessary rate.

5 These modes do not display graphics, and are provided for compatibility with BBC/Master series computers.

6 In these modes 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.

7 These are *gap modes*, where the colour of the gaps is not necessarily the same as the text background.

8 These modes 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.

9 This mode is not available in RISC OS 3 (version 3.00). It provides a double-sized display suitable for use by visually impaired people. Unfortunately some applications may not provide correct displays when used with this mode.

Other notes

Mode 32 has not been defined.

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.

In 256 colour modes, there are some restrictions on the control of the colours. Only 64 base colours may be selected; 4 levels of tinting turn the base colours into 256 shades. Also, the selection from the colour palette of 4096 shades is only possible in groups of 16.

Banks of screen memory

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 other banks may be accessed using OS_Bytes 112 - 113.

Effect of the mode command

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.

Getting information on a mode

The current screen mode may be read using OS_Byte 135 (page 1-670).

The size of the screen in a given mode can be determined by reading VDU variables XWindLimit, YWindLimit, XEigFactor, YEigFactor.

Example

```
VDU 22,7                    Select Teletext mode
```

VDU 23

Miscellaneous commands

Syntax

`VDU 23,`*`command`*`,`*`n1`*`,`*`n2`*`,`*`n3`*`,`*`n4`*`,`*`n5`*`,`*`n6`*`,`*`n7`*`,`*`n8`*

Parameters

`command`	the command to perform
`n1`, *`n2`*, *`n3`*, *`n4`*, *`n5`*, *`n6`*, *`n7`*, *`n8`*	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 BASIC VDU statements, as in the example below.

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 control 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 text 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 text cursor on the screen depending on the value of *`mode`*:

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

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

Example

`VDU 23,1,3|` *makes the cursor flash*

VDU 23,2-5

Define ECF pattern and colours

Syntax

`VDU 23,`*`pattern_no+1`*`,`*`n1`*`,`*`n2`*`,`*`n3`*`,`*`n4`*`,`*`n5`*`,`*`n6`*`,`*`n7`*`,`*`n8`*

Parameters

`pattern_no+1`	number of pattern to set (1 - 4) plus one
`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 section entitled *Application Notes* on page 1-770.

In any 256 colour mode, each parameter refers to the colour of each line. Use the colour byte as described by VDU 19 (page 1-588).

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 – which does the plotting – on page 1-628, and OS_Byte 163 – which sets the dot-dash repeat length – on page 1-675).

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 (as set by VDU 23,16)
5	scroll in negative x direction (as set by VDU 23,16)
6	scroll in positive y direction (as set by VDU 23,16)
7	scroll in negative y direction (as set by VDU 23,16)

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:

duration = 0	sets a steady flash colour 1
duration ≠ 0	sets the duration

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 (see page 1-649).

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:

duration = 0	sets a steady flash colour 2
duration ≠ 0	sets the duration

This command is equivalent to OS_Byte 10 (page 1-651).

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

Other 2 colour modes

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

4 colour modes

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

16 colour modes

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

256 colour modes

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,`*`pattern`*`,`*`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

The pattern parameter determines which colour pattern is set:

pattern	Sets colour pattern
12	1
13	2
14	3
15	4

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.

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 (page 1-588).

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) XOR 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 up
```

VDU 23,17,0-3

Set the tint for a colour

Syntax

`VDU 23,17,`*`action`*`,`*`tint`*`,0,0,0,0,0,0`

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 amount of white tint given to a colour in the 256-colour modes. The action determines which colour's tint 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

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

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)

For more details, see the section entitled *256-colour modes* on page 1-558.

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 pixel of the pattern coincides with the pixel at the specified point.

The origin is restored to the default after a mode change.

OS_SetECFOrigin (page 1-745) 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. Bit 1 of the flags 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 for compatibility with earlier operating systems. You must not use them. See the chapter entitled *The Font Manager* on page 3-411 for further details.

VDU 23,27

This call is provided by the Sprite Manager. See the chapter entitled *Sprites* on page 1-773 for further 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 (page 1-702).

You should not use this call in programs that might be run under the desktop, as your redefinitions may affect other programs. If you must use this call, ensure you only redefine characters that are normally unused.

Note that the desktop redefines some characters for its own use, and you must not redefine these yourself. To determine which characters are normally unused, view the entire system font under the desktop (the !Chars application is ideal for this):

- In RISC OS 2, the unused characters are those remaining from the underlined string 'These·characters·are·not·defined'
- In later versions of RISC OS, the unused characters are those represented as small hexadecimal numbers

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 (page 1-730) 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 end points, 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 (page 1-730).

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 (page 1-730).

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

Position 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 (page 1-668).

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.

Service Calls

Service_ModeChange (Service Call &46)

Mode change

On entry

R1 = &46 (reason code)
R2 = mode number
R3 = monitor type

On exit

All registers preserved (do not claim)

Use

This call is made whenever a mode change has taken place. It is made for the benefit of modules which may want to re-read some VDU variables to keep a consistent view of the world. Neither RISC OS 2 nor RISC OS 3 (version 3.00) pass the mode number and monitor type.

You should not claim this service call; there is nothing a module can do to prevent the mode change from taking place.

Service_PreModeChange (Service Call &4D)

Mode change

On entry

R1 = &4D (reason code)
R2 = selected mode (before possible translation)

On exit

Case 1

R1 preserved
R2 preserved

This is the normal action for a module which does not want to interfere

Case 2

R1 = 0 (service claimed)
R0 = 0

This implies that the module does not want the mode change to take place, and has taken an alternative action.

Case 3

R1 = 0 (service claimed)
R0 pointer to an error block

This implies that the module does not want the mode change to take place, and wishes to return the error pointed to by R0.

Case 4

R1 preserved
R2 = new mode

This implies that the module wants to substitute a mode for the specified mode. This is not a very good way of doing it, as other modules further down the chain will be offered the service with this new mode. The Service_ModeTranslation mechanism described below should be used by modules providing new monitor types.

Use

This service call is now of little use, as better alternatives are available.

Service_ModeExtension (Service Call &50)

Allow soft modes

On entry

R1 = &50 (reason code)
R2 = mode number that information is requested for
R3 = monitor type (or -1 for don't care)

On exit

All registers preserved (if not claimed)

If claimed:
R1 = 0
R2 preserved
R3 = pointer to VIDC list
R4 = pointer to workspace list

Use

This service call is issued when information is needed on a particular mode: for example on a mode change, or when mode variables are read. In RISC OS it is possible to load modules which provide additional screen modes and additional monitor types; such modules must claim this call if they recognise the passed mode and monitor type, and return the information.

Under RISC OS 3 (version 3.10) and later this service call is no longer issued for mode/monitor combinations that RISC OS itself already supports.

If writing a module providing soft modes, the mode number you use must fit this scheme:

Modes	Use
0 - 63	Reserved for use by RISC OS
64 - 95	Reserved for third party applications
96 - 127	Reserved for user defined modes

Mode numbers in the range 64 - 95 are allocated by Acorn.

Likewise, monitor types are allocated by Acorn. There are no monitor types pre-reserved for general use by users.

VIDC list: format 0

The returned VIDC list consists of a series of words. The first word specifies the format of the list, so this can be altered to cope with new hardware such as new versions of VIDC. There are currently two different formats. The first is:

Offset	Value
0	0 (format of list)
4	VIDC base mode
8	VIDC parameter
12	VIDC parameter
…	…
n	–1

The VIDC base mode is the number of an existing operating system screen mode which is used to determine the values of VIDC registers not explicitly mentioned in the list. The VIDC parameters are in the form that would be written to the hardware: ie the top 6 bits specify which register is programmed, and the remainder specify the value to be programmed in that register.

However, bits 6 and 7 of the control register should be set to 0, as these will be modified by RISC OS to take the configured sync and the *TV interlace setting into account. Similarly the vertical parameters for border start, display start, display end and border end are modified by RISC OS to take the *TV vertical offset into account.

VIDC parameters below &80000000 are ignored, since these correspond to palette registers (determined by the workspace base mode) and sound registers (not part of the display system).

VIDC list: format 1

On older machines, the VIDC clock is fixed at 24MHz, and the pixel rate is only determined by VIDC's internal dividers, as specified in bits 0 and 1 of the Control Register (VIDC address &E0). Newer machines have extra hardware to allow the selection of different VIDC clocks, and to determine the polarity of the sync lines. VIDC uses its clock together with a set of internal dividers to provide a range of pixel rates. For example, the A540 hardware provides the following pixel rates:

24000 kHz, 25175 kHz, 36000 kHz with a multiplier of 2/2
16000 kHz, 16783 kHz, 24000 kHz with a multiplier of 2/3
12000 kHz, 12587 kHz, 18000 kHz with a multiplier of 1/2
8000 kHz, 8392 kHz, 12000 kHz with a multiplier of 1/3

The second format of VIDC list was introduced to support these features. It is similar to format 0 (see above), but adds *extended parameters*:

Offset	Value
0	1 (format of list)
4	VIDC base mode
8	VIDC parameter
12	VIDC parameter
…	
n	–1
n+4	Extended parameter
n+8	Extended parameter
…	
m	–1

Extended parameters are of the form:

(0 « 24) + (pixel rate in kHz)

This will override the settings of bits 0 and 1 of a Control Register specifier in the main body of the list. If no pixel rate is specified, then the VIDC clock is set to 24MHz, and the settings of the divider in the Control Register are used as normal.

If the pixel rate specified is not achievable with the hardware on the machine, the nearest available pixel rate is used. When specifying a pixel rate for a hi-res-mono display, the pixel rate specified should be the actual pixel rate divided by 4, ie 24000 not 96000.

or:

(1 « 24) + (sync polarity)

where the sync polarity is defined as follows:

Bits	Meaning
0	0 ⇒ HSync +ve (as on a standard Archimedes), 1 ⇒ HSync –ve
1	0 ⇒ VSync +ve (as on a standard Archimedes), 1 ⇒ VSync –ve
2 - 23	reserved; must be zero

If no sync polarity is specified, a default of 0 is used (ie the same as a normal Archimedes).

or, from RISC OS 3 (version 3.10) onwards:

(2 « 24) + (true VIDC clock rate in kHz)

This is intended to be used in systems where the clock rate fed to VIDC is under the control of some external device, rather than being selected by the clock select latch. (For example, on the portable machine, the LCD ASIC feeds either 8MHz or 16MHz into VIDC when LCD modes are selected).

The values programmed into the clock select latch and the VIDC divider are still determined either from the control register specifier or a pixel rate specifier assuming the same range of clock speeds as on the A540; but the VIDC clock rate specifier is used to determine the video memory rate, which in turn determines the VIDC FIFO Request

Pointer values (bits 4 and 5 of the VIDC control register). The VIDC clock rate specifier is also stored in VDU variable VIDCClockSpeed (&AC), which is used by the SoundDMA module to determine the VIDC Sound Frequency Register value.

Workspace list

All values are words in the workspace list; its format is:

Offset	Value
0	0 (indicates format of list)
4	Workspace base mode
8	Mode variable index
12	Mode variable value
16	Mode variable index
20	Mode variable value
…	…
n	-1

The workspace base mode is the number of an existing operating system screen mode which is used to determine the values of mode variables not explicitly mentioned in the list. The mode variable indices are the same as for SWI OS_ReadModeVariable.

General notes

Modules can provide their own palette programming routines, including setting of the default palette, by claiming PaletteV. For more details see *PaletteV* on page 1-105, and *Service_ModeChanging* on page 1-648. This feature is not available under RISC OS 2, nor under RISC OS 3 (version 3.00); for these you should choose a workspace base mode which has an appropriate palette already set.

When the service is received, the module should check that R2 contains a mode that it knows about and that R3 holds a monitor type that is suitable for that mode. If not, the service should be passed on. If R3 holds -1 then the MOS is making a general enquiry about that mode (eg to determine the attributes of a sprite defined in that mode) so the module should only check R2.

Note that it is possible for a mode to have two or more different sets of VIDC parameters for different monitor types, but the workspace parameters **must** be the same, as the mode number is used as an identifier in sprites and in calls such as OS_ReadModeVariable.

Service_ModeTranslation (Service Call &51)

Translate modes for unknown monitor types

On entry

R1 = &51 (reason code)
R2 = mode number that requires translation
R3 = monitor type

On exit

All registers preserved (if not claimed)

If claimed:
R1 = 0
R2 = substitute mode
R3 preserved

Use

This service is offered during a call to OS_CheckModeValid or a screen mode change, if the selected mode is not available with the current monitor type (this having been ascertained by offering Service_ModeExtension) and the monitor type is not one known to the MOS (ie not in the range 0 - 3 for RISC OS 2.00, or 0 - 4 otherwise).

If the monitor type passed in R3 is known to the module, then the module should discover what the attributes of the mode in R2 are (by calling OS_ReadModeVariable) and then choose a mode which is suitable for this monitor type and is closest in attributes to the selected mode. This mode number should be returned in R2.

Service_MonitorLeadTranslation (Service Call &76)

Translate monitor lead ID

On entry

R1 = &76 (reason code)
R2 = monitor lead ID (see below)

On exit

If monitor lead ID is recognised, then the module should set:

R1 = 0 (claim service)
R3 = default screen mode number to use on this type of monitor
R4 = monitor type number to use (as used in *Configure MonitorType)
R5 = sync type to use on this type of monitor
(0 ⇒ separate syncs, 1 ⇒ composite sync)

All other registers must be preserved.

If the monitor lead ID is not recognised, the module should preserve all registers.

Use

This service call is issued if SWI OS_ReadSysInfo is called with R0=1 and any of the configured Mode, MonitorType or Sync are set to Auto.

The monitor connector provides 4 ID pins, ID0-ID3, each of which may be connected to 0v, +5v or to the Hsync pin. The monitor lead ID represents the state of these 4 ID pins by 4 2-bit fields, with ID0 in bits 0 and 1, ID1 in bits 2 and 3, ID2 in bits 4 and 5, and ID3 in bits 6 and 7. The meaning of each field is as follows:

Value	Meaning
0	ID pin is tied to 0v
1	ID pin is tied to +5v
2	ID pin is tied to Hsync
3	Indeterminate – either the state is fluctuating, or the machine is not capable of reading the ID

If your module recognises the monitor lead ID, you should claim the service call and return a monitor type, sync type, and default screen mode to use. For example, the Portable module (which is only present on portables) recognises the ID 1111 returned when no lead is connected, and sets the monitor type to 5 (LCD), the sync to 0, and the default mode to 27.

If the service is not claimed, then RISC OS checks the monitor lead ID against the following list of recognised IDs, and adopts these defaults:

ID0	ID1	ID2	ID3	Monitor type	Sync type	Default mode
H	1	1	X	0 (TV standard)	1 (composite)	12
1	1	H	X	1 (Multisync)	1 (composite)	27
1	0	1	X	3 (Mono VGA)	0 (separate)	27
0	1	1	X	3 (Colour VGA)	0 (separate)	27
0	1	0	X	4 (Super VGA)	0 (separate)	27

where the values in the ID columns have the following meanings:

Value	Meaning
0	ID pin must be tied to 0v
1	ID pin must be tied to +5v
H	ID pin must be tied to Hsync
X	Value of ID pin is immaterial

If RISC OS still does not recognise the monitor lead ID, it assumes it to be a TV standard monitor, and uses those settings (ie composite sync, default mode 12).

This service call is not issued by RISC OS 2. Furthermore, older machines such as the A300 and A400 series, and the A3000, cannot detect the sense of the monitor lead IDs.

Service_ModeChanging (Service Call &89)

Mode change

On entry

R1 = &89 (reason code)
R2 = mode number
R3 = monitor type

On exit

All registers preserved (do not claim)

Use

This service call is issued during a mode change, after Service_PreReset has been issued and a mode change is inevitable, but before it has actually happened. It is intended for use by modules that wish to claim PaletteV on particular combinations of mode number and monitor type. They should claim or release PaletteV depending on whether or not their module recognises the combination of mode number and monitor type. This allows modules providing extended palettes and the like to intercept RISC OS's palette programming.

For example, it is used in this way by the Portable module when an LCD screen mode is selected.

For more details, see the documentation of *PaletteV* on page 1-105.

This service call is not issued by RISC OS 2, nor by RISC OS 3 (version 3.00).

SWI Calls

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 (page 1-682).

Related SWIs

OS_Byte 10 (page 1-651), OS_Byte 195 (page 1-682)

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 (page 1-649), OS_Byte 194 (page 1-680)

Related vectors

ByteV

OS_Byte 19 (SWI &06)

Wait for vertical sync

On entry

R0 = 19

On exit

R0 preserved
R1, 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, you can 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 (page 1-660), OS_Byte 113 (page 1-662)

Related vectors

ByteV

OS_Byte 20 (SWI &06)

Reset font definitions

On entry

R0 = 20

On exit

R0 preserved
R1, 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 anyway not redefine characters in the range 32 - 127, since they all 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 (page 1-656)

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, 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 (page 1-654)

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 (page 1-710)

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 (page 1-696)

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 (page 1-698)

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 (page 1-660), OS_Byte 113 (page 1-662)

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 (page 1-677)

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 (page 1-730), OS_ReadModeVariable (page 1-736)

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 = status, or preserved
R2 = status, or 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 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

OS_Byte 134 (page 1-668)

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

R 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

OS_Byte 9 (page 1-649), OS_Byte 10 (page 1-651), OS_Byte 194 (page 1-680), OS_Byte 195 (page 1-682)

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 call reads the duration of the second colour, as set by OS_Byte 10.

The value read is only a record of the current duration. You must not attempt to use this call to write the value; doing so would merely change the stored value, without making any actual change. To change the duration you must instead call OS_Byte 10. This call is only included for backwards compatibility.

Related SWIs

OS_Byte 10 (page 1-651)

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.

The value read is only a record of the current duration. You must not attempt to use this call to write the value; doing so would merely change the stored value, without making any actual change. To change the duration you must instead call OS_Byte 9. This call is only included for backwards compatibility.

Related SWIs

OS_Byte 9 (page 1-649)

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 (page 1-686), OS_Byte 213 (page 1-688), OS_Byte 214 (page 1-690)

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 (page 1-684), OS_Byte 213 (page 1-688), OS_Byte 214 (page 1-690)

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 (in range 0 - 255)
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.

The frequency must be an old-style BBC frequency: ie in the range 0 - 255.

Related SWIs

OS_Byte 211 (page 1-684), OS_Byte 212 (page 1-686), OS_Byte 214 (page 1-690)

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 (page 1-684), OS_Byte 212 (page 1-686), OS_Byte 213 (page 1-688)

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 compatibility 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 call reads the current VDU driver screen bank number, as set by OS_Byte 112.

The value read is only a record of the current VDU driver screen bank. You must not attempt to use this call to write the value; doing so would merely change the stored value, without making any actual change. To change the duration you must instead call OS_Byte 112. This call is only included for backwards compatibility.

Related SWIs

OS_Byte 112 (page 1-660)

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 call reads the current display screen bank number, as set by OS_Byte 113.

The value read is only a record of the current display screen bank. You must not attempt to use this call to write the value; doing so would merely change the stored value, without making any actual change. To change the duration you must instead call OS_Byte 113. This call is only included for backwards compatibility.

Related SWIs

OS_Byte 113 (page 1-662)

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. In 256 colour modes this call ignores the tint, and returns a value in the range 0 - 63. Consequently you should always use OS_ReadPoint (page 1-734) in preference, since it returns both the logical colour and tint.

If the colour is returned as &FF then either:

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

This call is provided for backwards compatibility only.

Related SWIs

OS_ReadPoint (page 1-734)

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+8 = 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 (page 1-728). The OS_Word is provided for compatibility only.

Related SWIs

OS_ReadPalette (page 1-728)

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, 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 (see page 1-588).

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 (page 1-730). The OS_Word is provided for compatibility only.

Related SWIs

OS_ReadVduVariables (page 1-730)

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, R1 preserved

Interrupts

Interrupts are enabled (in RISC OS 2, the 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-resolution mono screens.

Related SWIs

OS_Byte 106 (page 1-658)

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	All treated as signed 16-bit values, relative to screen origin at the time the command is issued
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	

On exit

R0, 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 (treated as a signed 8-bit value)
    R1+2 = y multiplier (treated as a signed 8-bit value)

On exit

R0, 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 pointer move in the opposite direction from usual.

Both multipliers default to the configured MouseStep value. The factory defaults for this have varied between versions of RISC OS:

RISC OS version	MouseStep	Mouse movement per screen width
2	1	15cm
3 (version 3.00)	2	7.5cm
3 (version 3.10)	3	5cm

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, 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.

With RISC OS 2.00 this call generates an undefined instruction trap due to a stack mismatch. This was fixed in RISC OS 2.01.

Related SWIs

None

Related vectors

WordV

OS_Word 21,5 (SWI &07)

Set pointer position

On entry

R0 = 21 (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, 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 reads 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.

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, 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 (page 1-660), OS_Byte 113 (page 1-662)

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 (page 1-446)

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

RISC OS 3 (version 3.10) and later versions no longer return values in the range 0 - 15. Instead they always return 16 for BBC colours 0 - 7, and 17 and 18 for BBC flashing colours 8 - 15.

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, 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 page 1-736) with the following additions:

Name	No.	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
VIDCClockSpeed	172	VIDC clock speed in kHz (eg 24000 ⇒ 24 MHz) – not available in RISC OS 2.00

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 (page 1-736)

Related vectors

None

OS_ReadPoint (SWI &32)

Read the colour of a point

On entry

R0 = x coordinate
R1 = y coordinate

On exit

R0, 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 (page 1-588) 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, 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	No.	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-resolution mono mode = 1 Hi-resolution mono mode Bit 5 = 0 VDU characters are normal height = 1 VDU characters are double height Bit 6 = 0 hardware 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	Offset in bytes from a point on a pixel row to the same point on the pixel row below. On current hardware 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. You must not assume this will always be the case.
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 (page 1-730)

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 (page 1-741)

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 (page 1-739)

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:
    if R0 = –1 then mode is non-existent:
        R1 = mode that would be used, or –2 if unable to select alternative mode
    if R0 = – 2 then not enough memory:
        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_CheckModeValid determines whether you can change to a given mode and return with the carry bit appropriately set. If the mode you're checking is not 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 entitled *Memory Management* on page 1-343 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 - 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, 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 system information

On entry

R0 = reason code

On exit

R0 - R4 depend on reason code

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 reads various system information, depending on the reason code passed in R0. Current reason codes are:

R0	Meaning	Page
0	Read configured screen size in bytes	1-748
1	Read configured Mode, MonitorType, and Sync	1-749
2	Read presence of chips, and unique machine ID	1-750
3	Read features mask for 82C710 chip family	1-751

Related SWIs

None

Related vectors

None

OS_ReadSysInfo 0 (SWI &58)

Read the configured screen size in bytes

On entry

R0 = 0 (reason code)

On exit

R0 = amount of configured screen memory, in bytes

Use

This call reads the configured screen size in bytes (which is that used after the next hard reset). See **Configure ScreenSize* on page 1-764.

OS_ReadSysInfo 1 (SWI &58)

Read the configured Mode/WimpMode, MonitorType, and Sync

On entry

R0 = 1 (reason code)

On exit

R0 = configured Mode/WimpMode
R1 = configured MonitorType
R2 = configured Sync

Use

This call reads the configured Mode/WimpMode (which, from RISC OS 3 onwards, are identical), MonitorType, and Sync. See **Configure Mode* on page 1-757, **Configure WimpMode* on page 3-273, **Configure MonitorType* on page 1-759, and **Configure Sync* on page 1-766.

If any of the above have been configured to ‘Auto’, then the appropriate value for the attached monitor is returned.

This call is not available under RISC OS 2.

OS_ReadSysInfo 2 (SWI &58)

Read the presence of various chips, and unique machine ID

On entry

R0 = 2 (reason code)

On exit

R0 = IOEB ASIC presence flag
    0 ⇒ absent
    1 ⇒ present (type 1)
R1 = 82C710 (or similar) presence flag
    0 ⇒ absent
    1 ⇒ present
R2 = LCD ASIC presence flag
    0 ⇒ absent
    1 ⇒ present (type 1)
R3 = word 0 of unique machine ID, or 0 if unavailable
R4 = word 1 of unique machine ID, or 0 if unavailable

Use

This call checks for the presence of various chips, returning flags. It also reads the unique machine ID if a suitable chip is fitted to the computer; if none is, then the call returns an ID of zero.

Flag values not shown above are reserved for future hardware platforms that may have versions of the chips which are not backwards compatible.

This call is not available under RISC OS 2.

OS_ReadSysInfo 3 (SWI &58)

Read features mask for 82C710 chip family

On entry

R0 = 3 (reason code)

On exit

R0 = 82C710/82C711 basic features mask (see below)
R1 = 82C710/82C711 extra features mask (reserved for upwards compatible additional functionality)
R2-R4 reserved for future expansion

Use

The 82C710 family of chips are composed of several sub-units. Future chips in the family may have some sub-units which are incompatible with earlier versions, while leaving the functionality of other sub-units unchanged. This call returns a features mask, sub-fields within which show the 'compatibility level' of each sub-unit. Differing values of a sub-field indicate incompatible versions of the corresponding sub-unit. A sub-field of zero indicates that the sub-unit is not present. The values for the 82C710 and 82C711 are:

Bits	Sub-unit	82C710	82C711
0 - 3	IDE hard disc interface	1	1
4 - 7	floppy disc interface	1	1
8 - 11	parallel port	1	1
12 - 15	1st serial port	1	1
16 - 19	2nd serial port	0	1
20 - 23	chip configuration	1	2
24 - 31	reserved	0	0

(The 710 only supports a single serial port. Obviously the 710 and 711 have differing chip configurations.)

If a sub-unit gains additional backwards-compatible functionality in future versions of the chip, this will be indicated by having bits set in the value returned in R1. Information on extra sub-units will be accommodated in the remaining bits of R0, or in R2-R4.

This call is not available under RISC OS 2, nor under RISC OS 3 (version 3.00).

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 for disabled, 1 for enabled)
R1 = pointer 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

OS_SetColour (SWI &61)

Sets the foreground or background graphics colours

On entry

R0 = flags:

bits 0 - 3	graphics plotting action (see below)
bit 4	set ⇒ alter background, clear ⇒ alter foreground
bit 5	set ⇒ R1 = pattern data, clear ⇒ R1 = colour number
bits 6 - 31	reserved (must be zero)

R1 = colour number (if R0 bit 5 is clear),
or pointer to eight words of pattern data (if R0 bit 5 is set)

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the foreground or background graphics colours.

You can obtain the colour number to use from ColourTrans_ReturnColourNumber (page 3-352). You can supply an eight word pattern block to use giant ECFs instead of solid colours. With this call you can define a giant pattern for both the foreground and background colours, whereas the VDU drivers only allow you to set a single pattern for both (see VDU 23,2-5 on page 1-602).

The graphics plotting action passed in bits 0 - 3 of R0 is 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

Related SWIs

ColourTrans_ReturnColourNumber (page 3-352)

Related vectors

None

*Commands

*Configure Loud

Sets the configured volume for the beep to be loud

Syntax

```
*Configure Loud
```

Parameters

None

Use

*Configure Loud sets the configured volume for the beep to be loud.

The change takes effect on the next reset.

Related commands

*Configure Quiet

Related SWIs

OS_Byte 212 (page 1-686)

Related vectors

None

*Configure Mode

Sets the configured screen mode used

Syntax

`*Configure Mode` *`screen_mode`*`|Auto`

Parameter

`screen_mode`	the display mode that the computer should use after a power-on or hard reset, and when entering or leaving the desktop
`Auto`	automatic setting of appropriate mode using monitor lead

Use

*Configure Mode sets the configured screen mode used by the machine when it is first switched on, or after a hard reset, and when entering or leaving the desktop. It is identical to the command *Configure WimpMode; the two commands alter the same value in CMOS RAM.

You can also set a value of Auto (not available in RISC OS 2). More recent Acorn computers can sense the type of monitor lead connected, and hence set an appropriate mode. If no lead can be sensed, either because none is present or because the computer is of an older design, the mode defaults to mode 12.

Under RISC OS 2, this command only sets the configured screen mode used for the command line; *Configure WimpMode sets the configured screen mode used for the Desktop.

Example

`*Configure Mode 27` *selects VGA mode with 16 colours*

Related commands

*Configure WimpMode, VDU 22

Related SWIs

None

Related vectors

None

*Configure MonitorType

Sets the configured monitor type

Syntax

`*Configure MonitorType` *n*`|Auto`

Parameters

n	0 to 5
`Auto`	automatic sensing of monitor type using monitor lead

Use

*Configure MonitorType sets the configured monitor type that is connected to the computer. The values of *n* correspond to the following monitors:

n	Monitor	
0	50Hz TV standard colour or monochrome monitor	
1	Multiscan monitor	
2	Hi-resolution 64Hz monochrome monitor	
3	VGA-type monitor	
4	Super-VGA-type monitor	(not available in RISC OS 2)
5	LCD (liquid crystal display)	(only available on portables)

You can also set a value of Auto (not available in RISC OS 2). More recent Acorn computers can sense the type of monitor lead connected, and hence set the monitor type. If no lead can be sensed, either because none is present or because the computer is of an older design, the monitor type defaults to 0, save for portables, which default to 5 (an LCD).

You can also configure the monitor type by holding down the corresponding key from the numeric keypad while the computer is switched on.

Example

```
*Configure MonitorType 3          configure VGA-type monitor
```

Related commands

VDU 22

Related SWIs

None

Related vectors

None

*Configure MouseStep

Sets the configured value for how fast the pointer moves as you move the mouse

Syntax

```
*Configure MouseStep n
```

Parameters

n a number between 1 and 127

Use

*Configure MouseStep sets the configured value for how fast the pointer moves as you move the mouse. Useful values of n 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.

You can also use OS_Word 21,2 to set the mouse step dynamically.

Example

```
*Configure MouseStep 3          select a fast speed
```

Related commands

None

Related SWIs

OS_Word 21,2 (page 1-714)

Related vectors

None

*Configure NoScroll

Sets the configured scrolling so the screen does not scroll upwards at the end of a line

Syntax

```
*Configure NoScroll
```

Parameters

None

Use

*Configure NoScroll sets the configured scrolling so that a newline is not generated when a character is printed at the end of a line. The default value 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 for details of how to set this option dynamically.

Related commands

*Configure Scroll

Related SWIs

None

Related vectors

None

*Configure Quiet

Sets the configured volume for the beep to half its loudest volume.

Syntax

```
*Configure Quiet
```

Parameters

None

Use

*Configure Quiet sets the configured volume for the beep to half its loudest volume.

The change takes effect on the next reset.

Related commands

*Configure Loud

Related SWIs

OS_Byte 212 (page 1-686)

Related vectors

None

*Configure ScreenSize

Sets the configured amount of memory reserved for screen display

Syntax

```
*Configure ScreenSize mK|n
```

Parameters

*m*K	number of kilobytes of memory reserved
n	number of pages of memory reserved; n ≤ 127

Use

*Configure ScreenSize sets the configured amount of memory reserved for screen display. The default value is 80Kbytes on a 0.5Mbyte machine, and 160Kbytes for all other machines.

You can also use OS_ChangeDynamicArea (page 1-384) to alter the screen memory allocation dynamically. For more information, refer to the chapter entitled *Memory Management*.

You cannot configure more than 480Kbytes of screen memory, due to limitations in the MEMC1 and MEMC1a memory controllers.

Example

```
*Configure ScreenSize 160K        reserve 160Kbytes for screen display
```

Related commands

None

Related SWIs

OS_ChangeDynamicArea (page 1-384)

Related vectors

None

*Configure Scroll

Sets the configured scrolling so the screen scrolls upwards at the end of a line

Syntax

```
*Configure Scroll
```

Parameters

None

Use

*Configure Scroll sets the configured scrolling so that a newline is generated automatically when a character is printed at the end of a line. This is the default value.

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 for details of how to set dynamically this option.

Related commands

*Configure NoScroll

Related SWIs

None

Related vectors

None

*Configure Sync

Sets the configured type of synchronisation for vertical sync output

Syntax

```
*Configure Sync 0|1|Auto
```

Parameters

`0`	vertical sync
`1`	composite sync
`Auto`	automatic sensing of required synchronisation type using monitor lead

Use

*Configure Sync sets the configured type of synchronisation that is provided on the vertical sync output of the video connector. This may be vertical sync (parameter of 0) or composite sync (parameter of 1).

You can also set a value of Auto (not available in RISC OS 2). More recent Acorn computers can sense the type of monitor lead connected, and hence set the required synchronisation type. If no lead can be sensed, either because none is present or because the computer is of an older design, the synchronisation type defaults to 1 (composite sync).

Example

```
*Configure Sync 1
```

Related commands

None

Related SWIs

None

Related vectors

None

*Configure TV

Sets the configured vertical screen alignment and screen interlace

Syntax

```
*Configure TV [vert_align[[,]interlace]]
```

Parameters

vert_align	adjusts the vertical screen alignment 0 to 3 lines up (values of 0-3 respectively), or 1 to 4 lines down (values of 255-252 respectively)
interlace	switches screen interlace on (with a value of 0), or off (with a value of 1)

Use

*Configure TV sets the configured vertical screen alignment and screen interlace. 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

*Shadow

Sets which bank of screen memory is used on subsequent mode changes

Syntax

```
*Shadow [0|1]
```

Parameters

`0` or `1` or nothing

Use

*Shadow sets which bank of screen memory is used on subsequent changes to the screen mode. It controls two banks of screen memory: the normal bank (bank 1, known as the *non-shadow* bank), and an alternate bank (bank 2, known as the *shadow* bank).

If you give either no parameter, or a parameter of 0, the shadow bank is used on the next mode change. If you give a parameter of 1, the non-shadow bank is used on the next mode change.

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. For example, to use shadow memory in screen mode 8 (a mode which requires 40Kbytes), at least 80Kbytes of screen memory must be available.

This command is provided for backwards compatibility only, since there is no useful benefit in using twice as much screen memory.

Example

```
*Shadow 1
```

Related commands

*Configure ScreenSize

Related SWIs

None

Related vectors

None

*TV

Adjusts the vertical screen alignment and screen interlace

Syntax

`*TV [`*`vert_align`*`[[,]`*`interlace`*`]]`

Parameters

`vert_align`	adjusts the vertical screen alignment 0 to 3 lines up (values of 0-3 respectively), or 1 to 4 lines down (values of 255-252 respectively)
`interlace`	switches screen interlace on (with a value of 0), or off (with a value of 1)

Use

*TV adjusts the vertical screen alignment and screen interlace.

The change takes effect on the next mode change.

Example

`*TV 3,0` *move the picture up 3 lines, and turn interlace on*

Related commands

*Configure TV

Related SWIs

None

Related vectors

None

Application Notes

Examples of ECF pattern use

This section gives some examples of how you might set ECF patterns using the VDU 23,2-5… commands.

In BBC/Master compatible mode

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
```

22 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, or transformed, 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*, from the Wimp's common sprite pool, or from any number of *user sprite areas*.

System sprite area

The system sprite area 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, 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

Format of a sprite area

The format of a sprite area is as follows:

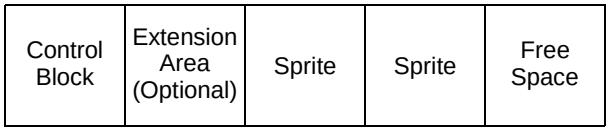


Figure 22.1 Format of a sprite area

The sprite area control block contains the following:

Bytes	Contents
0 - 3	Byte offset to last byte+1 (ie total size of sprite area)
4 - 7	Number of sprites in area
8 - 11	Byte offset to first sprite
12 - 15	Byte offset to first free word (ie byte after last sprite)
16…	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, save that word 1 of the control block is not saved. (There is no need to save this, as the total size of the sprite area is the size of the file).

Format of a sprite

The format of a sprite is as follows:

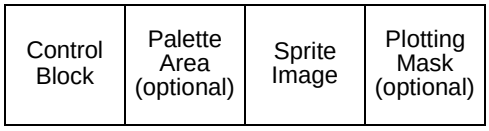


Figure 22.2 Format of a sprite

The Sprite Control Block contains the following:

Bytes	Content
0 - 3	Offset to next sprite
4 - 15	Sprite name, up to 12 characters with trailing zeroes
16 - 19	Width in words –1

20 - 23	Height in scan lines –1
24 - 27	First bit used (left end of row)
28 - 31	Last bit used (right end of row)
32 - 35	Offset to sprite image
36 - 39	Offset to transparency mask or offset to sprite image if no mask
40 - 43	Mode sprite was defined in (see page 1-779)
44…	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.

Each entry is two words long. These are the words returned from OS_ReadPalette. The format of these words is described with this SWI on page 1-728.

256 colour modes

256 colour modes may be an exception to this rule, because there are only 16 palette registers in VIDC. Most 256 colour sprites will have 16 palette entries; those created by *ScreenSave actually have 64 palette entries; some generated by programs will have a full 256 palette entries. The standard RISC OS display routines pass the last 16 entries to VIDC. For notes on using sprites with 256 entry palettes, see the section entitled *Using sprites with 256 entry palettes* on page 1-859.

Format of a sprite image

The format of a sprite image is as follows:

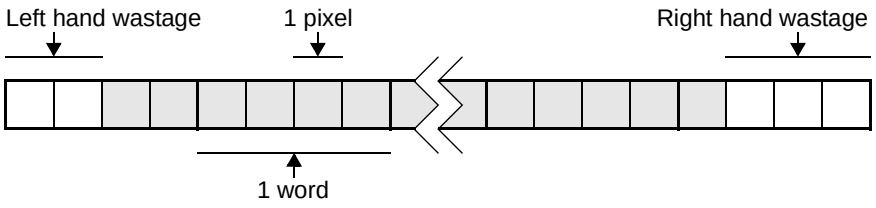


Figure 22.3 Format of a sprite image

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 OS_ReadVduVariables on page 1-730). 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).

Common parameters

Several kinds of parameter 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. Note however that direct pointers to sprites in a given area will become invalid if any sprite in that area changes size or is replaced.

Sprite modes

To make your sprites readable by old releases of RISC OS, we recommend you use the following mode numbers in a sprite (see the section entitled *Format of a sprite* on page 1-777):

	2 colours	4 colours	16 colours	256 colours
2 × 4 OS unit pixels	0	8	12	15
4 × 4 OS unit pixels	4	1	9	13
2 × 2 OS unit pixels	18	19	20	21

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 multiplier and divided by the divisor, ie:

x pixel size = x start size (in pixels) × x multiplier / x divisor
y pixel size = y start size (in pixels) × y multiplier / 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 section entitled *ECF patterns* on page 1-552 for a description of ECF patterns.

If the pointer is zero, then no scaling is performed; ie 1:1 scale.

The Wimp uses a similar system to provide mode independence. For details see *Wimp_ReadPixTrans* on page 3-205.

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 *n*th 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. For details see Wimp_SetPalette, Wimp_ReadPalette, and Wimp_SetColour from page 3-187 onwards.

The ColourTrans module provides facilities for translation table calculations. For more information refer to the chapter entitled *ColourTrans* on page 3-333.

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 with sprite pixel colour
1	OR colour on screen with sprite pixel colour
2	AND colour on screen with sprite pixel colour
3	exclusive OR colour on screen with sprite pixel colour
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND colour on screen with NOT of sprite pixel colour
7	OR colour on screen with NOT of sprite pixel colour
&08	If set, then use the mask; otherwise don't

Save area

When you switch output to a sprite or its mask using OS_SpriteOp 60 (page 1-838) or OS_SpriteOp 61 (page 1-840), you can save the VDU context in a save area. The save area you pass 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 required size of which you can obtain by calling OS_SpriteOp 62 (page 1-842). You cannot directly manipulate the contents, but for your reference the save area 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 system save area for the screen. Therefore, if you swap output to a sprite, perform some operations and swap back, it will not be necessary to allocate a save area.

When you first switch output to a sprite or mask your save area must have a zero in the first word; it is therefore ignored, and the VDU state set to the default for the mode in which the sprite is defined. When you switch output away from the sprite or mask, RISC OS saves the VDU state to the save area, setting the first word to a non-zero value. Hence when that save area is next passed to OS_SpriteOp 60 or 61, RISC OS recognises that it contains a VDU state and restores it.

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 (*page 1-794) or *SNew (page 1-856). To change the system sprite area size, you can call OS_ChangeDynamicArea (page 1-384); you can also change the configured size of this area (which is used on a hard reset) by calling *Configure SpriteSize (page 1-843), or – except under RISC OS 2 – by using the Configure application.

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 (page 1-852) or OS_SpriteOp 8 (page 1-793) 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 (page 1-798) 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 (page 1-847) and OS_SpriteOp 2 (page 1-791) will perform this operation. *ScreenLoad (page 1-846) and OS_SpriteOp 3 (page 1-792) 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 (page 1-854) and OS_SpriteOp 10 (page 1-795) will load a sprite file into an initialised sprite area and set up all the pointers within it. To save, *SSave (page 1-858) and OS_SpriteOp 12 (page 1-797) 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 (page 1-855) and OS_SpriteOp 11 (page 1-796) 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 (page 1-799) or 16 (page 1-801), or *SGet (page 1-851). Alternatively, you can create a blank sprite with OS_SpriteOp 15 (page 1-800) 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 (page 1-807) 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 (page 1-805) or *SCopy (page 1-845) can be used. They will return an error if the designated name already exists.
- To rename a sprite, OS_SpriteOp 26 (page 1-804) or *SRename (page 1-857) can be used. Again, the same error condition applies to existing destination names.
- To delete a sprite, its mask and palette, OS_SpriteOp 25 (page 1-803) or *SDelete (page 1-848) can be used. You can delete the mask of a sprite only, by calling OS_SpriteOp 30 (page 1-808). 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 (page 1-809) to insert a row
- OS_SpriteOp 32 (page 1-810) to delete a row
- OS_SpriteOp 45 (page 1-823) to insert a column
- OS_SpriteOp 46 (page 1-824) to delete a column
- OS_SpriteOp 57 (page 1-836) to insert or delete rows
- OS_SpriteOp 58 (page 1-836) to insert or delete columns

Axis flipping

A sprite can be flipped about its x or y axis. Flipping it about the x axis using OS_SpriteOp 33 (page 1-811) or *SFlipX (page 1-849) will make it appear upside down. Flipping it about the y axis with OS_SpriteOp 47 (page 1-825) or *SFlipY (page 1-850) 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 (page 1-832) 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 (page 1-813). No extra memory is used to do this.

Reading and altering pixels

To check the size of a sprite, OS_SpriteOp 40 (page 1-818) 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 (page 1-819) will return colour and tint for a given x and y coordinate in the sprite. To write a pixel colour, OS_SpriteOp 42 (page 1-820) 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 (page 1-821) will read a mask pixel and OS_SpriteOp 44 (page 1-822) 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 (page 1-838) will switch output to a sprite or to the screen. OS_SpriteOp 61 (page 1-840) 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 (page 1-842).

Plotting sprites

To plot a sprite on the screen, OS_SpriteOp 28 (page 1-806) and 34 (page 1-812) are the simplest to use. They plot the sprite at the current graphics cursor position, using the current GCOL action. OS_SpriteOp 48 (page 1-826) and 49 (page 1-827) are similar, but the coordinates and GCOL action are instead passed explicitly.

Scaled and transformed plotting

A sprite can be plotted at any magnification using OS_SpriteOp 50 (page 1-828) and 52 (page 1-830).

Like these SWIs, OS_SpriteOp 53 (page 1-831) 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 (page 1-829) will paint a character onto the screen using scale factors.

OS_SpriteOp 55 and 56 (page 1-833) will plot a sprite or mask with a linear transformation, such as a shear, stretch or reflection.

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 (see page 1-628). *x* and *y* are in OS coordinates.

Service Calls

Service_SwitchingOutputToSprite (Service Call &72)

Output switched to sprite, mask or screen

On entry

R1 = &72 (reason code)
R2 = value passed in R0 to SpriteOp that caused output to switch
R3 = value passed in R1 to SpriteOp that caused output to switch
R4 = value passed in R2 to SpriteOp that caused output to switch
R5 = value passed in R3 to SpriteOp that caused output to switch

On exit

All registers preserved

Use

Issued when output is switched from and to a sprite immediately after the output is switched.

This service call should not be claimed.

SWI Calls

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 (RISC OS 2.00); SWI is re-entrant (RISC OS 2.01 or later)

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	Meaning	Page
2	Screen save	1-791
3	Screen load	1-792
8	Read area control block	1-793
9	Initialise sprite area	1-794
10	Load sprite file	1-795

R0	Meaning	Page
11	Merge sprite file	1-796
12	Save sprite file	1-797
13	Return name	1-798
14	Get sprite	1-799
15	Create sprite	1-800
16	Get sprite from user co-ordinates	1-801
24	Select sprite	1-802
25	Delete sprite	1-803
26	Rename sprite	1-804
27	Copy sprite	1-805
28	Put sprite	1-806
29	Create mask	1-807
30	Remove mask	1-808
31	Insert row	1-809
32	Delete row	1-810
33	Flip about x axis	1-811
34	Put sprite at user coordinates	1-812
35 †	Append sprite	1-813
36 †	Set pointer shape	1-815
37 †	Create/remove palette	1-817
40	Read sprite information	1-818
41	Read pixel colour	1-819
42	Write pixel colour	1-820
43	Read pixel mask	1-821
44	Write pixel mask	1-822
45	Insert column	1-823
46	Delete column	1-824
47	Flip about y axis	1-825
48	Plot sprite mask	1-826
49	Plot mask at user coordinates	1-827
50 †	Plot mask scaled	1-828
51 †	Paint character scaled	1-829
52 †	Put sprite scaled	1-830
53 †	Put sprite grey scaled	1-831
54	Remove lefthand wastage	1-832
55 †	Plot mask transformed	1-833

R0	Meaning	Page
56 †	Put sprite transformed	1-833
57 †	Insert/delete rows	1-836
58 †	Insert/delete columns	1-836
60	Switch output to sprite	1-838
61	Switch output to mask	1-840
62	Read save area size	1-842

For details of each of these reason codes, see below.

Note that the reason codes marked with a dagger are provided by the SpriteExtend module, which must be loaded for them to work.

Related SWIs

None

Related vectors

SpriteV

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, R2, 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 (page 1-792)

Related vectors

SpriteV

OS_SpriteOp 3 (SWI &2E)

Screen load

On entry

R0 = 3
R2 = pointer to pathname

On exit

R0, 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 (page 1-791)

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, 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 – for a description see the section entitled *Common parameters* on page 1-779.

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, 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 10 (SWI &2E)

Load sprite file

On entry

R0 = 10
R1 = pointer to control block of sprite area
R2 = pointer to pathname

On exit

R0 - 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 sprite area must be initialised before you call this SWI.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 11 (SWI &2E)

Merge sprite file

On entry

R0 = 11
R1 = pointer to control block of sprite area
R2 = pointer to pathname

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 12 (SWI &2E)

Save sprite file

On entry

R0 = 12
R1 = pointer to control block of sprite area
R2 = pointer to pathname

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 13 (SWI &2E)

Return name

On entry

R0 = 13
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 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

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, R1 preserved
R2 = address of sprite (if in user sprite area)
R3 preserved

Use

This defines the identified sprite 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 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 1-779. 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 (page 1-801)

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 - R6 preserved

Use

This creates a blank sprite of a given size.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 1-779. 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, R1 preserved
R2 = address of sprite (if in user sprite area)
R3 - 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 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 1-779. 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 (page 1-799)

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, 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.

The returned address only remains valid until the next SpriteOp which may rearrange the sprite area, such as OS_SpriteOp 11 (merge sprite file) or OS_SpriteOp 25 (delete sprite).

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 1-779.

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 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

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 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

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 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 28 (SWI &2E)

Put sprite

On entry

R0 = 28
R1 = pointer to control block of sprite area
R2 = sprite pointer
R5 = plot action (see page 1-781)

On exit

R0 - R2, 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.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 48 (page 1-826)

Related vectors

SpriteV

OS_SpriteOp 29 (SWI &2E)

Create mask

On entry

R0 = 29
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 30 (page 1-808)

Related vectors

SpriteV

OS_SpriteOp 30 (SWI &2E)

Remove mask

On entry

R0 = 30
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 29 (page 1-807)

Related vectors

SpriteV

OS_SpriteOp 31 (SWI &2E)

Insert row

On entry

R0 = 31
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = row number

On exit

R0 - 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, or to transparent if the sprite has a mask. 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 32 (page 1-810), OS_SpriteOp 45 (page 1-823), OS_SpriteOp 46 (page 1-824)

Related vectors

SpriteV

OS_SpriteOp 32 (SWI &2E)

Delete row

On entry

R0 = 32
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = row number

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 31 (page 1-809), OS_SpriteOp 45 (page 1-823), OS_SpriteOp 46 (page 1-824)

Related vectors

SpriteV

OS_SpriteOp 33 (SWI &2E)

Flip about x axis

On entry

R0 = 33
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 47 (page 1-825)

Related vectors

SpriteV

OS_SpriteOp 34 (SWI &2E)

Put sprite at user coordinates

On entry

R0 = 34
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate at which to plot
R4 = y coordinate at which to plot
R5 = plot action (see page 1-781)

On exit

R0 - R5 preserved

Use

This plots a sprite at the external coordinates supplied, using the plot action supplied in R5.

No check is made that the plotted sprite's screen mode is compatible with the current screen mode; if they are not compatible you will get a display that may not be particularly useful.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 35 (SWI &2E)

Append sprite

On entry

R0 = 35
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

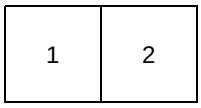
On exit

R0 - R4 preserved

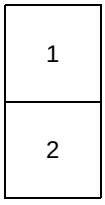
Use

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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 36 (SWI &2E)

Set pointer shape

On entry

R0 = 36
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 - 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
7 - 31	reserved; must be zero

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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_Word 21,0 (page 1-710), Wimp_SetPointerShape (page 3-163)

Related vectors

SpriteV

OS_SpriteOp 37 (SWI &2E)

Create/remove palette

On entry

R0 = 37
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = sub-reason code:
–1 ⇒ read current palette size
0 ⇒ remove palette from sprite
otherwise ⇒ create palette in sprite, extended to 256 entries if bit 31 set

On exit

R0 - R2 preserved
R3 = size of palette or 0 if none (if R3 = –1 on entry); else preserved
R4 = pointer to palette or 0 if none (if R3 = –1 on entry)
R5 = mode (if R3 = –1 on entry)

Use

This call creates a palette, removes a palette, or finds the size of the palette associated with a given sprite.

If you add or remove a sprite's palette when output is switched to the sprite you will invalidate the current display pointers In such cases you should switch output away from the sprite, modify the palette, and then switch output back to the sprite.

To create 256 entry palettes you must set bit 31 of R3 on entry. This facility is not available in RISC OS 2.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 40 (SWI &2E)

Read sprite information

On entry

R0 = 40
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 41 (SWI &2E)

Read pixel colour

On entry

R0 = 41
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate (in pixels)
R4 = y coordinate (in pixels)

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 42 (page 1-820)

Related vectors

SpriteV

OS_SpriteOp 42 (SWI &2E)

Write pixel colour

On entry

R0 = 42
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate (in pixels)
R4 = y coordinate (in pixels)
R5 = colour
R6 = tint

On exit

R0 - R6 preserved

Use

Given x and y coordinates (in pixels relative to the bottom left of the sprite definition), and colour and tint in R5 and R6, this call sets the colour of the pixel at that position.

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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 41 (page 1-819)

Related vectors

SpriteV

OS_SpriteOp 43 (SWI &2E)

Read pixel mask

On entry

R0 = 43
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate (in pixels)
R4 = y coordinate (in pixels)

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 44 (page 1-822)

Related vectors

SpriteV

OS_SpriteOp 44 (SWI &2E)

Write pixel mask

On entry

R0 = 44
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate (in pixels)
R4 = y coordinate (in pixels)
R5 = mask status (0 = transparent, 1 = solid)

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 43 (page 1-821)

Related vectors

SpriteV

OS_SpriteOp 45 (SWI &2E)

Insert column

On entry

R0 = 45
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = column number

On exit

R0 - 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 zero 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 31 (page 1-809), OS_SpriteOp 32 (page 1-810), OS_SpriteOp 46 (page 1-824)

Related vectors

SpriteV

OS_SpriteOp 46 (SWI &2E)

Delete column

On entry

R0 = 46
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = column number

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 31 (page 1-809), OS_SpriteOp 32 (page 1-810), OS_SpriteOp 45 (page 1-823)

Related vectors

SpriteV

OS_SpriteOp 47 (SWI &2E)

Flip about y axis

On entry

R0 = 47
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 33 (page 1-811)

Related vectors

SpriteV

OS_SpriteOp 48 (SWI &2E)

Plot sprite mask

On entry

R0 = 48
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - 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).

The plot action for this call is the same as that for normal graphics operations, rather than that used in plotting sprites.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 28 (page 1-806)

Related vectors

SpriteV

OS_SpriteOp 49 (SWI &2E)

Plot mask at user coordinates

On entry

R0 = 49
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate at which to plot
R4 = y coordinate at which to plot

On exit

R0 - R4 preserved

Use

This plots in the background colour and action through a sprite mask at the external coordinates supplied.

The plot action for this call is the same as that for normal graphics operations, rather than that used in plotting sprites.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 48 (page 1-826)

Related vectors

SpriteV

OS_SpriteOp 50 (SWI &2E)

Plot mask scaled

On entry

R0 = 50
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate at which to plot
R4 = y coordinate at which to plot
R6 = scale factors

On exit

R0 - R6 preserved

Use

A sprite mask is plotted on the screen, using the current background colour and action and the scaling factors provided.

The plot action for this call is the same as that for normal graphics operations, rather than that used in plotting sprites.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 51 (SWI &2E)

Paint character scaled

On entry

R0 = 51
R1 = character code
R3 = x coordinate at which to plot
R4 = y coordinate at which to plot
R6 = scale factors

On exit

R0, R1, R3, R4, 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.

The plot action for this call is the same as that for normal graphics operations, rather than that used in plotting sprites.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 52 (SWI &2E)

Put sprite scaled

On entry

R0 = 52
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate at which to plot
R4 = y coordinate at which to plot
R5 = plot action (see page 1-781)
R6 = scale factors: 0 ⇒ no scaling (ie 1:1, sprite pixel to screen pixel)
R7 = pixel translation table: 0 ⇒ no translation

On exit

R0 - 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 factors specified by R6
- the pixel translation table pointed to by R7

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 53 (page 1-831)

Related vectors

SpriteV

OS_SpriteOp 53 (SWI &2E)

Put sprite grey scaled

On entry

R0 = 53
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate at which to plot
R4 = y coordinate at which to plot
R5 = 0
R6 = scale factors
R7 = pixel translation table

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 52 (page 1-830)

Related vectors

SpriteV

OS_SpriteOp 54 (SWI &2E)

Remove left hand wastage

On entry

R0 = 54
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - 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 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 60 (page 1-838), OS_SpriteOp 61 (page 1-840)

Related vectors

SpriteV

OS_SpriteOp 55 and 56 (SWI &2E)

Transformed sprite handling

On entry

R0 = 55 (PlotMaskTransformed) or 56 (PutSpriteTransformed)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = flag word:
    bit 0 set ⇒ R6 = pointer to destination coordinates, else matrix
    bit 1 set ⇒ R4 = pointer to source rectangle inside sprite
    bits 2-31 reserved (must be 0)
R4 = pointer to source rectangle coordinate block (if R3 bit 1 set):
    R4+0,4 = x0, y0: one corner in sprite (in pixels)
    R4+8,12 = x1, y1: second corner in sprite (in pixels)
R5 = GCOL action (for PutSpriteTransformed)
    +8 if mask is to be used
R6 = pointer to matrix (if R3 bit 0 clear):
    R6+0,4,8,12,16,20 = matrix (as for Draw module)
R6 = pointer to destination coordinate block (if R3 bit 0 set):
    R6+0,4 = X0,Y0 on screen (1/256th OS unit)
    R6+8,12 = X1,Y1 on screen (1/256th OS unit)
    R6+16,20 = X2,Y2 on screen (1/256th OS unit)
    R6+24,28 = X3,Y3 on screen (1/256th OS unit)
R7 = pointer to translation table (≤ 0 ⇒ none)

On exit

—

Use

This call is not available in RISC OS 2.

The source coordinates are inclusive at the bottom-left, and exclusive at the top-right. If no source rectangle is given, the default is x0 = 0, x1 = width of sprite (in pixels), and y0 = height of sprite (in pixels), y1 = 0. Note that the y coordinates are the reverse of what you might expect; this is only relevant when plotting to a destination parallelogram (see below).

When specifying a destination parallelogram, the source rectangle is mapped onto the destination as follows:

x0,y0	X0,Y0
x1,y0	X1,Y1
x1,y1	X2,Y2
x0,y1	X3,Y3

In future it may be possible to set the destination to an arbitrary quadrilateral, rather than a parallelogram. In order to reserve this possibility, the current version returns an error if the destination is not a parallelogram.

For PutSpriteTransformed, the sprite is plotted through its mask only if it both has one, and bit 3 of R5 is set. R5 is ignored for PlotMaskTransformed.

The SWI returns an error if any of R3 bits 2 - 31 are set, to ensure that these are left clear by software developers.

The SWI covers exactly those pixels on the screen that a call to Draw_Fill would produce for a rectangle of the same size with the same transformation matrix, where it is filling to half-way through the boundary.

When plotting using a destination parallelogram, the source rectangle must be entirely within the sprite. For plotting with a matrix, the source rectangle will be clipped to the sprite boundaries prior to transformation; any of the sprite's pixels outside the source rectangle will behave as if they had a transparent mask.

If the source rectangle (after clipping, if using a matrix) has no area, i.e. x0 = x1 OR y0 = y1 then an error will be generated, as it is not possible to choose a colour in which to fill the destination.

Note that the SWI does allow x0>x1 or y0>y1 or both. When plotting with a matrix there is no difference between x0 and x1 swapped, or y0 and y1 swapped, but when specifying a destination parallelogram the image will be reflected.

Due to the mechanism of the routine the accuracy is not absolute. The SWI will always cover the same area as a Draw filled path, but not necessarily with the right source pixel data from the sprite. The worst possible error (in a fraction of a source pixel) at one end of the plotted area is given by *destination width or height*/65536.

The table below gives more information on the maximum errors attainable:

Destination size	Worst possible error in source pixels
5	0.0000763
10	0.0001526
50	0.0007629
100	0.0015259
500	0.0076294
1000	0.0152588
5000	0.0762939
10000	0.1525879

(The largest output possible is 32767 pixels)

For example, when plotting a sprite to a destination width of 5000 pixels, the worst error possible in the position in the source rectangle of the final pixel plotted is about $1/13$ of a source pixel.

Note that if these errors (usually too small to notice) must be avoided then the sprite should be plotted in parts – perhaps by dividing the plotting into four areas.

Errors:

‘Attempt to set reserved flags’:

R0 bits 2 - 31 must be zero.

‘Source rectangle area zero’:

The area of the source rectangle must be non-zero, so the sprite routine(s) will have some valid colour with which to plot the output.

‘Source rectangle not inside sprite’:

The source rectangle must be totally inside the sprite.

‘SpriteExtend can only do linear transformations’:

The current version of the transformation routines can only perform linear transformations, and not any arbitrary distortion.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 57 and 58 (SWI &2E)

Insert/delete rows/columns from a sprite

On entry

R0 = 57 (InsertDeleteRows) or 58 (InsertDeleteColumns)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = row/column to start deletion at or to insert before
R4 = number of rows/columns to insert (if +ve) or delete (if -ve)

On exit

R0 - R4 preserved

Use

This call is not available in RISC OS 2.

For insertion R4 > 0, and R3 specifies the row or column to insert before. For a sprite of *n* rows × *m* columns the rows are numbered from 0 at the bottom to *n*–1 at the top, and columns from 0 at the left to *m*–1 at the top. Thus to insert rows/columns on the edges of the sprite:

R0	R3	Insertion point
57	0	bottom edge (ie before the first row)
	n	top edge (ie before the row beyond the last row)
58	0	left edge (ie before the first column)
	m	right edge (ie before the column beyond the last column)

The inserted rows/columns are set to colour 0. If the sprite has a mask then rows/columns are inserted into that as well, and the inserted area is transparent.

For deletion R4 < 0, and R3 specifies the first row or column to be deleted. The rows/columns from R3 to (R3–R4–1) will be deleted. An error will be given if R3 or R4 are out of range for the sprite.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 60 (SWI &2E)

Switch output to sprite

On entry

R0 = 60
R1 = pointer to control block of sprite area
R2 = sprite pointer to switch to sprite, or 0 to switch to screen
R3 = pointer to save area, or 0 for no save area, or 1 for system save area

On exit

R0 preserved
R1 = previous value
R2 = previous value
R3 = previous value

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; alternatively it can be zero, in which case output is switched to the screen.

A save area is used to save graphics context. This call can be thought of as switching to a particular graphics context as well as switching output.

The save area's location (specified in R3) can have a number of values. If it is zero, then no save area will be used; you should avoid this if possible. If it is one, then the system save area is used; RISC OS uses this area to save the screen's graphics context, and you should not use this area yourself if you wish to preserve the screen's graphics context. Any other value of R3 is considered to be a pointer to a user specified save area, usually used to preserve a sprite's graphics context.

When you make this call, the current graphics state is saved to the current graphics save area, the first word of which is set to a non-zero value. The save area specified by R3 is then made the current save area. If its first word is non-zero, it is assumed to contain a graphics context, which is restored; if its first word is zero (or no save area is specified), then it is assumed to be a new save area, and the VDU state is instead initialised to suitable defaults for the mode in which the sprite was defined. Output is then switched, as specified by R2.

You can find the required size of a save area by calling OS_SpriteOp 62.

You must not poll the Wimp whilst output is switched to a sprite, as other applications will not expect this.

For more details of save areas, and examples of their use, see the section entitled *Save area* on page 1-781.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 61 (page 1-840), OS_SpriteOp 62 (page 1-842)

Related vectors

SpriteV

OS_SpriteOp 61 (SWI &2E)

Switch output to mask

On entry

R0 = 61
R1 = pointer to control block of sprite area
R2 = sprite pointer to switch to mask, or 0 to switch to screen
R3 = pointer to save area, or 0 for no save area, or 1 for system save area

On exit

R0 preserved
R1 = previous value
R2 = previous value
R3 = previous value

Use

This call can cause VDU calls to be sent to the screen memory, or to a sprite's mask.

See OS_SpriteOp 60 for a general description of how this call works.

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. Hence 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 255.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 60 (page 1-838), OS_SpriteOp 62 (page 1-842)

Related vectors

SpriteV

OS_SpriteOp 62 (SWI &2E)

Read save area size

On entry

R0 = 62
R1 = pointer to control block of sprite area
R2 = sprite pointer, or 0 for the screen

On exit

R0 - 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. This is a constant for a particular release of RISC OS, but may vary between versions. Remember that a save area must be word aligned.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 1-779.

Related SWIs

OS_SpriteOp 60 (page 1-838), OS_SpriteOp 61 (page 1-840)

Related vectors

SpriteV

*Commands

*Configure SpriteSize

Sets the configured amount of memory reserved for the system sprite area

Syntax

```
*Configure SpriteSize mK|n
```

Parameters

*m*K	number of kilobytes of memory reserved
n	number of pages of memory reserved; n ≤ 127

Use

*Configure SpriteSize sets the configured amount of memory reserved for the system sprite area. If you pass a parameter of 0, then no space is reserved for system sprites. The default value is one page of memory.

You can also use OS_ChangeDynamicArea (page 1-384) to alter dynamically the system sprite size. For more information, refer to the chapter entitled *Memory Management*.

The change takes effect on the next hard reset.

Example

```
*Configure SpriteSize 20K
```

Related commands

None

Related SWIs

None

Related vectors

None

*SChoose

Selects a sprite for use in subsequent sprite plotting operations

Syntax

`*SChoose` *`sprite_name`*

Parameters

`sprite_name`	name of a sprite in the system sprite area

Use

*SChoose selects a sprite from the system sprite area for use in subsequent sprite plotting operations. It is used in conjunction with VDU 25,232-239 operations. You should see the warning in the section entitled *VDU commands* on page 1-786 about using these obsolescent VDU calls.

The sprite name is not case-sensitive.

Example

```
*SChoose fish
```

Related commands

None

Related SWIs

OS_SpriteOp 24 (page 1-802)

Related vectors

SpriteV

*SCopy

Makes a copy of a sprite within the system sprite area

Syntax

`*SCopy` *`source_sprite_name dest_sprite_name`*

Parameters

`source_sprite_name`	name of source sprite in the system sprite area
`dest_sprite_name`	name of destination sprite (to be placed in the system sprite area)

Use

*SCopy makes a copy of the source sprite within the system sprite area, and renames it as the destination sprite. An error is generated if the destination sprite already exists.

Example

```
*SCopy acorn squirrel
```

Related commands

None

Related SWIs

OS_SpriteOp 27 (page 1-805)

Related vectors

SpriteV

*ScreenLoad

Loads the contents of a sprite file into the graphics window

Syntax

```
*ScreenLoad filename
```

Parameters

filename	a valid pathname, specifying a sprite file

Use

*ScreenLoad loads the contents of a sprite file (saved, for example, with the *ScreenSave command) into the graphics window, which is typically the whole screen.

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 (page 1-792)

Related vectors

SpriteV

*ScreenSave

Saves the contents of the graphics window and its palette to a file

Syntax

`*ScreenSave` *`filename`*

Parameters

`filename`	a valid pathname, specifying a file

Use

*ScreenSave saves the contents of the graphics window (typically the whole screen) and its palette to a file, which is saved as a sprite. The sprite file created will contain one sprite called ‘screendump’.

You can then load this file into Paint or Draw.

Example

```
*ScreenSave My.Pic
```

Related commands

*ScreenLoad

Related SWIs

OS_SpriteOp 2 (page 1-791)

Related vectors

SpriteV

*SDelete

Deletes one or more sprites from the system sprite area

Syntax

`*SDelete` *`sprite_name1`* [*`sprite_name2…`*]

Parameters

`sprite_name1`	name of a sprite in the system sprite area
`sprite_name2…`	optional extra sprites to delete

Use

*SDelete deletes one or more sprites from the system sprite area.

If an error occurs (such as a sprite not existing) *SDelete will stop immediately, and no further sprites will be deleted.

Example

```
*SDelete fish cake elephant
```

Related commands

None

Related SWIs

OS_SpriteOp 25 (page 1-803)

Related vectors

SpriteV

*SFlipX

Reflects a sprite in the system sprite area about its x axis

Syntax

`*SFlipX` *`sprite_name`*

Parameters

`sprite_name`	name of a sprite in the system sprite area

Use

*SFlipX reflects a sprite in the system sprite area about its x axis so it is upside down.

Example

```
*SFlipX sloth
```

Related commands

*SFlipY

Related SWIs

OS_SpriteOp 33 (page 1-811)

Related vectors

SpriteV

*SFlipY

Reflects a sprite in the system area about its y axis

Syntax

`*SFlipY` *`sprite_name`*

Parameters

`sprite_name`	name of a sprite in the system sprite area

Use

*SFlipY reflects a sprite in the system sprite area about its y axis so it faces in the opposite direction.

Example

```
*SFlipY sloth
```

Related commands

*SFlipX

Related SWIs

OS_SpriteOp 47 (page 1-825)

Related vectors

SpriteV

*SGet

Gets a sprite from the screen

Syntax

`*SGet` *`sprite_name`*

Parameters

`sprite_name`	name of new sprite in the system sprite area

Use

*SGet gets a sprite from a rectangular area of the screen, defined by the two most recent graphics positions (inclusive). It then saves this sprite in the system sprite area with the given name. If the sprite already exists, it is overwritten.

Any part of the designated area which lies outside the current graphics window is filled in the sprite with the current background colour.

Example

```
*SGet screenpart
```

Related commands

*ScreenSave

Related SWIs

OS_SpriteOp 14 (page 1-799)

Related vectors

SpriteV

*SInfo

Displays information on the system sprite workspace

Syntax

```
*SInfo
```

Parameters

None

Use

*SInfo displays information on the system sprite workspace. It 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 (page 1-793)

Related vectors

SpriteV

*SList

Lists the names of all the sprites in the system sprite area

Syntax

```
*SList
```

Parameters

None

Use

*SList lists the names of all the sprites in the system sprite area.

Example

```
*SList
!koala
!sloth
```

Related commands

None

Related SWIs

OS_SpriteOp 8 (page 1-793)

Related vectors

SpriteV

*SLoad

Loads a sprite file into the system sprite area

Syntax

`*SLoad` *`filename`*

Parameters

`filename`	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 (page 1-795)

Related vectors

SpriteV

*SMerge

Merges the sprites in a file with those in the system sprite area

Syntax

`*SMerge` *`filename`*

Parameters

`filename`	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 (page 1-796)

Related vectors

SpriteV

*SNew

Deletes all the sprites in the system sprite area

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 (page 1-794)

Related vectors

SpriteV

*SRename

Renames a sprite within the system sprite area

Syntax

```
*SRename old_sprite_name new_sprite_name
```

Parameters

old_sprite_name	name of a sprite in the system sprite area
new_sprite_name	new name of the sprite

Use

*SRename renames a sprite within the system sprite area. An error is generated if a sprite having the new name already exists.

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 (page 1-804)

Related vectors

SpriteV

*SSave

Saves the system sprite area as a sprite file

Syntax

`*SSave` *`filename`*

Parameters

`filename`	name of file to save

Use

*SSave saves all the sprites currently in the system sprite area as a sprite file. You can then load or merge the file later on.

Example

```
*SSave $.sprites.animals.koala
```

Related commands

*SLoad, *SMerge

Related SWIs

OS_SpriteOp 12 (page 1-797)

Related vectors

SpriteV

Application notes

Using sprites with 256 entry palettes

Introduction

By careful use of sprite operations and ColourTrans, sprites with 256 entry palettes can be created and displayed on all RISC OS machines. Of course, this ability to create and process a sprite with (say) 256 grey levels in it does not magically endow the hardware with the ability to display 256 grey levels! The display will be as close as possible (ie 16 grey levels with standard hardware), but will not be exact unless the machine's hardware has been extended using a product such as a graphics enhancer expansion card.

Format of a 256 entry palette sprite

A 256 entry palette sprite is precisely like a 16 entry palette sprite, save that there are 256 palette entries, each one consisting of a pair of 'palette entry' words of the form &BBGGRR00 as for ColourTrans. All bits of the entry are significant.

Creating a 256 entry palette sprite

If you use OS_SpriteOp 15 to create a sprite with a palette for a 256 colour mode, the palette will not have 256 entries. To create a sprite with the full 256 palette entries, the best thing to do is to create one with no palette, and then to add 2048 to the following entries:

Entry	Meaning
word 4 of Sprite Area control block	offset to first free word
word 1 of Sprite control block	offset to next sprite
word 9 of Sprite control block	offset to sprite image
word 10 of Sprite control block	offset to transparency mask

You should then write the 256 double words of palette entries starting at word 12 of the sprite, keeping both items in a pair identical:

```
SYS"OS_SpriteOp",&10f,ram%,name$,0,X,Y,spritemode
sptr%=ram%+ram%!8
pal%=sptr%+11*4
!(sptr%+8*4)+=2048
!(sptr%+9*4)+=2048
!sptr%+=2048
!(ram%+12)+=2048
FORZ%=0TO255:B%=palette!(Z%<<2)ANDNOT&FF
pal%!(Z%*8)=B%:pal%!(Z%*8+4)=B%
NEXT
```

Manipulating a 256 entry palette sprite

All sprite operations work on the 256 entry palette sprite. One can even switch output to it and generate 256 grey level output into it (with appropriate care over the GCOL and TINT values required by RISC OS). Sprite areas containing 256 entry palette sprites may be loaded, saved etc.

Testing to see if a sprite is a 256 entry palette sprite

After creation, the 256 entry palette sprite behaves just like all the others, so it is important you can distinguish it on the occasions when you need to (such as when displaying it on the screen).

A 256 entry palette sprite will have the lowest of words 9 and 10 of the sprite control block equal to 2048+44 (&82C). If you already know that the sprite has no transparency mask, then you can test only word 9.

Displaying a 256 entry palette sprite

The SWI ColourTrans_SelectTable takes a 64 entry palette sprite (and 2, 4 and 16) and returns a pixel translation table as needed for OS_SpriteOp 52. For a 256 entry sprite, one needs to build a similar pixel translation table directly. The following code will compute a pixel translation table for any sprite which hasn't a transparency mask:

```
IF sptr%!32=44 THEN
palptr%=0
ELSE
FOR grab%=0 TO 2048-8 STEP 8
paltemp%!(grab%>>1)=sptr%!(grab%+44)
NEXT
palptr%=paltemp%
ENDIF
FORQ%=0TO255:pixtrans%?Q%=Q%:NEXT
IFsptr%!32=44+2048 THEN
FORQ%=0TO255
SYS"ColourTrans_ReturnColourNumber",palptr%!(Q%<<2) TO
pixtrans%?Q%
NEXT
ELSE
SYS "ColourTrans_SelectTable",m,palptr%,-1,-1,pixtrans%
ENDIF
spx%=-1:FORQ%=0TO255:IFpixtrans%?Q%<>Q% spx%=pixtrans%
NEXT
```

`spx%` is equal either to –1 if no translation needs to be done (which speeds up OS_SpriteOp 52 a lot), or to pixtrans%; it is passed to OS_SpriteOp 52 in register 7.

Conclusion

The ability to store, process and display 256 entry palette sprites represents a small but useful gain for the RISC OS desktop. Common formats can be converted to sprites without any loss of information and meaningfully displayed. It's still a good idea to use the default desktop palette whenever possible.

23 Character Input

Introduction

The Character Input system can get characters from the computer's input devices. They can be any one of the following:

- the keyboard
- the serial port
- a file on any filing system

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

It provides a means of directing characters from the selected device to the program that requests them. It can also hold them, waiting until the program is ready to take them.

For details of input to Wimp applications, see the chapter entitled *The Window Manager* on page 3-3.

Overview

Before you read this chapter, you should have read the chapter entitled *Character Output* on page 1-503. In many ways, character input and output are one entity, which has been logically split in this manual. So there are some things which are mentioned there and not here that apply to both chapters.

Like character output, a stream system is used by character input. Here, you can select from one of three streams; keyboard, serial and file. Only one stream can be selected at once otherwise data coming from two places would get jumbled. Direct control of devices is available, especially in the case of the keyboard.

Streams

Any program taking input from the stream system doesn't have to know where characters are coming from. Most programs don't since it will not affect the way they run.

OS_ReadC

The core of the input stream is OS_ReadC (page 1-880) which gets a single character from the currently selected input stream. It is in turn called by many other SWIs, OS_ReadLine (page 1-941) for example. This device independence makes programs much easier to write.

Buffers

Like character output, all input streams are buffered. Input devices are asynchronous to programs and must have their characters stored in a temporary place in memory until required. A good example of buffering in use is a terminal emulator program. It waits until something appears at the serial input buffer, then sends it to the VDU. At the same time, it waits until something appears in the keyboard buffer and sends it to the serial output buffer. Because of the buffering of inputs and outputs, the program can do all this at its own pace.

Keyboard

The keyboard is the most used part of character input, and its driver the most complex. In principle it is simple enough, but many features are changeable and key presses can be looked at in a number of ways.

Keyboard handlers

The keyboard driver is actually two sections. One, which is fixed, handles the keyboard interrupt and low-level control. It feeds the raw code onto the second part, the keyboard handler.

The keyboard handler converts the keycode into an ASCII form, with extensions for special characters. This can be replaced by a custom version if required.

Basic operation

At a basic level, the keyboard works like this:

1 One or more keys are pressed, which cause an interrupt.

2 The keyboard driver gets a raw key number from the keyboard.

3 The raw key number is passed to the keyboard handler, where it is converted into a form more like the program expects. This can be:

- an ASCII character.
- a non-ASCII character, such as a function key or arrow.
- a special key, such as Escape or Break that must be acted on immediately.

4 Apart from some special keys, this character is then stored in the keyboard buffer.

When a program wants a character from the input stream (in this example, the keyboard):

- When called by a program, the stream system gets the first character from the keyboard buffer (or waits if there is none there).
- Return the character to the program or perform the appropriate action if it is a function key, arrow, etc.

Advanced features

Also, there are a number of extra operations that the keyboard driver can perform:

- The interpretation of function keys, arrow keys and the numeric keypad can all be changed to various modes.
- The auto-repeat of keys can be adjusted, both the initial delay and the rate of repeat.
- The keyboard can be scanned directly, rather than going through any buffering.
- The keyboard handler can even be completely replaced with a custom handler.

About 30 SWIs and six * Commands exist purely for keyboard control. The section entitled *Technical Details* on page 1-869 covers how they work together.

Reset, Break and Escape

These three terms can become very confused, especially so when talking about the keyboard versus a program's view of the keyboard driver.

Reset

Reset is a unique key. Unlike all others it does not send a key code to the keyboard driver. It is connected to a separate line on the keyboard connector and physically resets the computer. This cannot be stopped by a program. When a reset occurs, some parts of the system are initialised.

Pressing the **Reset** switch alone causes a 'soft' reset. This resets your machine, restarting RISC OS. You will lose any unsaved work.

You can get other types of Reset options by holding down certain keys whilst you press the Reset switch:

- Holding down the **Ctrl** key causes a 'hard reset'. This is more severe than a soft reset (but still doesn't reset your machine as thoroughly as switching it off, then on again).
- Holding down the **Shift** key reverses the action of the configured boot option. If there **is** a boot file set to run, it is **not** run. If there is a boot file **not** set to run, it **is** run.
- Holding down ***(on numeric keypad)-Reset** causes the Command Line to be entered, rather than the configured language (such as the Desktop or BASIC).

You can combine the effects of these keys; for example pressing ***(on numeric keypad)-Ctrl-Shift-Reset** on a machine configured to auto-boot would cause a hard reset, after which the Command Line would be entered, and the boot file would not be run.

BBC/Master users note that Reset is what used to be called Break on those machines.

Break

Break is a key. You can separately configure Break, Shift-Break, Ctrl-Break and Ctrl-Shift-Break to cause a reset, an escape condition or do nothing.

By default, pressing the **Break** key (to the right of the twelve function keys) on its own acts like pressing the Escape key; for instance it may interrupt a program. However, if you press it whilst holding down any of the keys that affect the Reset switch it acts like the Reset switch, except that it does not reset the computer's hardware. For example:

- Pressing **Shift-Break** causes a soft restart of RISC OS, reversing the normal auto-boot behaviour.
- Pressing **Ctrl-Break** causes a hard restart of RISC OS, which is more severe in its effects.

The computer has a Break key as well as a Reset switch so that applications such as emulators can respond differently to them. For example, 65Host uses Break to reset the emulated computer, while Reset still resets the RISC OS computer itself.

Escape

Escape is a way of the user sending a signal to a program or its runtime environment. From a program's point of view, we talk about an escape condition. This can be caused by an escape key or the program itself.

By default, the key that causes an escape condition is Escape. RISC OS can be configured so that the escape key is any key on the keyboard.

When an escape condition occurs, RISC OS will call the escape handler of the program or the language environment. See the description of handlers in the section entitled *Handlers* on page 1-292. The escape handler or running program should then clear the escape condition and act in an appropriate way. Note that it is perfectly valid for a program to ignore an escape condition as long as it is cleared.

The escape event can also be enabled. This is called in place of the escape handler. (See the chapter entitled *Events* on page 1-147.)

Serial port

A character which comes into the serial port interrupts the computer. It is then placed into the serial input buffer, if it is enabled. RISC OS can be configured so that serial input is ignored.

The computer can be set up so that input coming in from the serial port is treated exactly as if it had come from the keyboard. This means that the escape character and function key codes will be recognised.

If characters come in the serial port too quickly to be processed, then the serial input buffer would become full. After this point, data would be lost. To solve this problem, the serial driver will notify the sender to stop transmitting before it gets full. From a program's point of view, this all happens invisibly.

Calls that are specific to the serial port, whether they refer to input or output (eg those to set the baud rate, or to explicitly send/receive a character from/to the serial port), are gathered together in the chapter entitled *Serial device* on page 2-445.

*Exec

*Exec is the opposite of spooling, which is used in character output. *Exec makes a file the current input stream. Keyboard and serial input is ignored.

A SWI is provided to allow the Exec file to be changed or stopped under program control.

Technical Details

Events

There are a number of events associated with the character input system. In particular:

- input buffer has become full
- character placed in input buffer
- a key has been pressed/released
- serial error has occurred
- escape condition detected

See the chapter entitled *Events* on page 1-147 for more details of these events.

Streams

OS_ReadC (page 1-880) is the core of the input stream system. It is called by many SWIs and it uses one of the three streams as an input source. The stream that it uses can be controlled by OS_Byte 2 (page 1-882) for keyboard and serial port. To use the third stream, the file, then *Exec (page 2-167) or OS_Byte 198 (page 1-908) can be used. OS_Byte 177 (page 1-902) can be used to read the setting of the last OS_Byte 2.

OS_ReadC is also responsible for handling cursor-editing during input.

OS_ReadLine

OS_ReadLine (page 1-941), and its obsolete equivalent OS_Word 0, will read a line of input from the current input stream. It copes with the deleting of characters or the whole line. Thus, a single call which returns a simple string to the program allows the user much flexibility.

Keyboard

When a key is pressed (or released), a code unique to that key is transmitted to the computer through the keyboard connector cable. This code is read into some hardware, which causes an interrupt to occur. The keyboard driver responds to this interrupt by reading the keycode, and passing it on to the keyboard handler for further processing.

At this stage, a key press/release event may be generated, which you can handle as required. Also, at this level mouse button presses look exactly the same as any other key press. It is only when the mouse button presses reach the keyboard handler that they are recognised as such, and RISC OS is informed that the mouse button state has changed.

Keyboard buffer

The keyboard buffer is often termed a type-ahead buffer, as it enables the user to type commands ahead of the program being ready for them. You must not assume it to be any particular length.

Disabling buffering

OS_Byte 201 (page 1-912) will stop the keyboard handler from putting any characters it gets into the keyboard buffer. This means that most keyboard reading calls will not work. Where this function is useful is if you want a program to insert codes directly into the buffer without any of the user's key strokes appearing in the middle of them.

Keyboard status

If the key pressed (or released) is one of the shifting keys (Shift, Ctrl or Alt) or one of the locking keys (Caps Lock, Num Lock or Scroll Lock) is pressed, then the key handler just makes a note of this fact by updating its status information. Normally this doesn't cause any character to be inserted into the keyboard buffer; although the Alt key can in combination with the numeric keypad – see *Table D: Character sets* on page 4-569.

OS_Byte 202 (page 1-914) allows reading and writing of the keyboard status byte. This is a bitfield that represents the state of Shift, Ctrl, Alt and all the Lock keys. If it is written and any of the Lock keys with LEDs are changed, then this will not be reflected in the LEDs. OS_Byte 118 (page 1-891) must be called to do this.

The next time any key goes down or up, then the Shift, Alt and Ctrl states will reflect their real position and the LEDs will be updated to their current status.

You can use *Configure Caps, NoCaps and ShCaps (page 1-947 onwards) to set the default Caps Lock key state.

Scanning keys

Scanning refers to being able to get the low level key codes without the buffering and interpretation that is placed on keys by the higher level routines. The internal key number returned is not the code that the keyboard itself sends the computer. This is translated to a standard internal key number that maintains compatibility with BBC/Master series keyboard codes.

There are three OS_Bytes that can scan the keyboard. OS_Byte 121 (page 1-892) can scan a particular key or a range or keys. Like this call, OS_Byte 122 (page 1-894) can scan a fixed range of keys, all but the Shift, Alt, Ctrl and mouse keys. OS_Byte 129 (page 1-899) can scan a particular key, like OS_Byte 121. It can also read a key with a time limit. This is discussed later.

Key handler

The character stored in the keyboard buffer is derived from a table in the key handler, which maps the low level key codes into buffer codes, using the state of the various shifting and locking keys to alter the character if appropriate. In addition, the key-press is recorded in a ‘last key pressed’ location. This is to enable auto-repeating keys to be implemented, as described below.

For the standard keys, eg the letters, digits, punctuation marks etc, the buffer code is the ASCII code of the symbol. Thus when the code comes to be removed from the keyboard buffer (by OS_ReadC, for example), it is returned directly to the user. The other keys, such as the function keys and cursor keys, are entered as top-bit set characters, in the range &80 - &FF.

Custom key handler

The SWI OS_InstallKeyHandler (page 1-945) allows replacing the module that decodes key numbers into ASCII. It is outside the scope of this manual to discuss this procedure in depth.

Read with time limit

OS_Byte 129 (page 1-899) supports two operations, one of which, low level keyboard scanning, was discussed in the earlier section on scanning keys.

The other allows reading a character from the keyboard buffer within a time limit. This is useful in cases where a program waits for a response for a time, and if none is entered, continues. It can be used in a situation where the keyboard buffer needs to be checked periodically, but the program doesn’t wish to be trapped waiting in OS_ReadC for a character to be entered. To achieve this, this call would be used with no waiting time, so if no characters are available in the buffer, then the program can continue.

Tab key

OS_Byte 219 (page 1-918) reads or modifies the code inserted into the keyboard buffer when the Tab key is pressed (the default is 9). If the value specified is in the range &80 to &FF, then the value to be inserted is modified by the state of the Shift and Ctrl keys, in a similar fashion to the function keys.

Auto-repeat

The auto-repeat of keys has two aspects: the delay before the key starts repeating, and the rate of repeating. The delay can be read and changed with OS_Byte 196 (page 1-904), or changed with OS_Byte 11 (page 1-886). The rate can be read and changed with OS_Byte 197 (page 1-906), or changed with OS_Byte 12 (page 1-888). Both are adjustable from 1 to 255 centiseconds. Auto-repeat can also be disabled.

The delay and rate can be set up using *Configure Delay and Repeat (page 1-948 onwards), which use the same parameter as the appropriate OS_Bytes.

Arrow and Copy keys

In a default system, these keys are used for on-screen editing. The arrows move a cursor and Copy copies the character that it is on to the second cursor.

OS_Byte 237 (page 1-929) reads how the cursor keys are interpreted. As well as the default editing state, they can be in two other modes. In one, the keys return characters in the range 135 to 139. In the other, they act as function keys, and can be treated as all the other function keys.

Although you can also use OS_Byte 237 to change this state, OS_Byte 4 (page 1-884) is the preferred way of doing so.

Numeric keypad

There is a base value for the numeric keypad. A key on the numeric keypad adds an offset to this to get the character that is placed in the keyboard buffer. The offset of each key is such that the default base value of 48 will give each key the ASCII value of the character on the key.

This base value can be changed with OS_Byte 238 (page 1-931). See the documentation on this call for details of the offsets of each key.

Shift and Ctrl can alter the value returned from the keypad. By default, this feature is disabled, but you can enable it with OS_Byte 254 (page 1-937).

Interpreting characters &80 - &FF

When referring to function keys, we are talking about two separate things. There are the keys, many discussed earlier, that generate buffer codes in the range &80 to &FF. Then there is the interpretation placed upon these buffer codes by RISC OS as it reads them from the buffer.

Interpreting these keys as function keys is only one way of using them. OS_Bytes 221 - 228 (page 1-922) allow control over how buffer codes from &80 to &FF are interpreted by RISC OS. Each OS_Byte handles a group of 16 characters. Each group can be configured so that its characters are:

- interpreted as function keys
- preceded by a NULL (ASCII 0)
- offset by any number from 3 - &FF
- discarded

Function keys

If a character is read from the keyboard buffer and is in a group that is configured as function keys, then a special action is taken by the keyboard handler. First of all, it looks up the value of the Key$*n* system variable which corresponds to the function key. The function key number is the lower nibble of the character. Thus, if the character is &81, the variable read is Key$1.

The variable refers to a string, which is copied into the function key buffer. If the string was a null string (the function key wasn't set), then RISC OS continues, removing the next character from the input buffer.

Otherwise, the first character is removed from the function key buffer and returned to the calling program. Characters read from this buffer are returned without interpretation in any way.

Subsequent calls to OS_ReadC and OS_Byte 129 spot that a function key is being read, and remove characters from the function key buffer instead of looking in the input buffer. This continues until the last character has been read from the buffer. Input then reverts to the normal input buffer.

OS_Byte 216 (page 1-916) is used to see how much of a function key string remains to be read from the function key buffer. It can also change this value, to terminate for instance, but must be used with care.

Setting and clearing

To set a function key, a number of commands can be called:

- *Key *n string*
- *Set Key$*n string*
- *SetMacro Key$*n expression*. This is passed through OS_GSTrans when it is copied to the function key buffer. This is interesting because it means that the string generated by a function key can change every time it is used.

To reset one or more function keys, there is also a variety of commands that can be used:

- *Key *n* will reset function key *n*
- *Unset Key$*n* will also reset function key *n*
- *Unset Key$* will reset all function keys
- OS_Byte 18 will also reset all function keys

Reset, Break and Escape

Reset

When you press the Reset button, then the RISC OS ROM is paged into the bottom of memory and performs certain housekeeping actions. It then pages itself out and restarts the system.

A soft reset distinguishes itself from a hard reset in a matter of degree. A hard reset will initialise far more things in the system. A soft reset, for instance, will not change the settings for PrinterType and the printer ignore character, nor will it reinitialise relocatable modules. It will, however, reset vectors that have been claimed.

OS_Byte 200 (page 1-910) sets whether a reset will act as described above or will cause a complete memory clear. This makes it a power-on reset. If this is used, then all things kept in memory will be lost and settings restored to the defaults stored in CMOS RAM. This command should be used with discretion because of its powerful effects.

OS_Byte 253 (page 1-935) can be used to see what kind of reset the last one was.

Break

Break is configurable with OS_Byte 247 (page 1-933). This sets how Break, Shift Break, Ctrl Break and Ctrl Shift Break act. They can each be set to cause a reset or an escape or have no effect. A reset caused by the break key does not page the ROM into the bottom of memory (as one caused by the Reset button does); instead, it just jumps to the correct location in the ROM.

Escape

The diagram below illustrates how all the calls in the escape system work together. A description of this interaction follows the diagram.

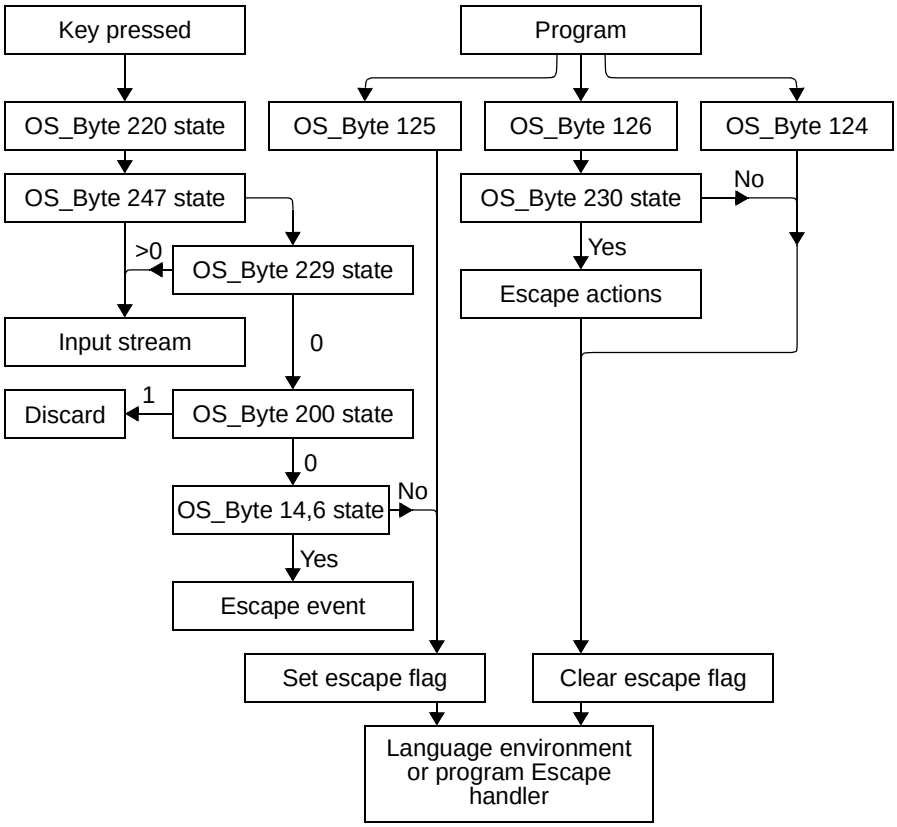


Figure 23.1 Interaction of calls in the escape system

Causing escape

An escape condition can be caused by a key or under program control. By default, the escape key is Escape. OS_Byte 220 (page 1-920) can read or alter which key will cause an escape condition. OS_Byte 247 (page 1-933) can alter the Break key (or Shift and Ctrl modifiers of it) so that it causes an escape condition. Thus, it is possible to have two escape keys on the keyboard, and this is indeed the default state.

Under program control, OS_Byte 125 (page 1-896) can force an escape condition to occur. Note that it will not generate an event, but the escape handler is called.

OS_ReadEscapeState (page 1-943) can check whether an escape condition has occurred. It can be called at any time, even from within interrupts.

Disabling escape

OS_Byte 229 (page 1-925) controls recognition of this escape character. It can disable the effect of the escape character and allow it to pass through the input stream unaltered. OS_Byte 200 (page 1-910) can disable all escape conditions apart from those caused by OS_Byte 125. In this case, any escape characters would be discarded.

OS_Byte 14,6 (page 1-152) controls whether the escape event is enabled or not. If the escape event is enabled, then it will be called and not the escape handler.

After an escape

OS_Byte 126 (page 1-897) will acknowledge an escape condition and call the escape handler to clear up. OS_Byte 124 (page 1-895) will clear an escape condition without calling the escape handler.

OS_Byte 230 (page 1-927) controls whether the normal effects of an escape occur or not when it is acknowledged. These include flushing buffers, closing the Exec file, terminating any sounds and so on.

Serial device

The serial device is provided as a DeviceFS (*Device Filing System*) device. For full details, see the chapter entitled *DeviceFS* on page 2-429, and the chapter entitled *Serial device* on page 2-445. The latter chapter also contains all calls that are specific to the serial port, whether they refer to input or output (eg those to set the baud rate, or to explicitly send/receive a character from/to the serial port).

*Exec

There are two ways of causing a file to be made the input stream. The simplest is to use *Exec (page 2-167), which will open the specified file and attach it as the input stream. For more control, OS_Byte 198 (page 1-908) does what *Exec does, and can also terminate the Exec stream at any time or change to another file.

Internal key numbers

The diagrams below show the BBC/Master compatible internal key numbers generated by different keyboards.

General points

Some keys generate two numbers; this is for compatibility with Master computers. Codes 0 - 2 are generated by both of the Shift, Ctrl and Alt keys respectively; this is useful for testing if either or both of the left and right hand keys have been pressed.

Furthermore, the mouse buttons generate internal key numbers:

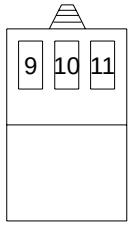


Figure 23.2 Internal key numbers for mouse

Standard Archimedes keyboard

The standard Archimedes keyboard generates these internal key numbers:

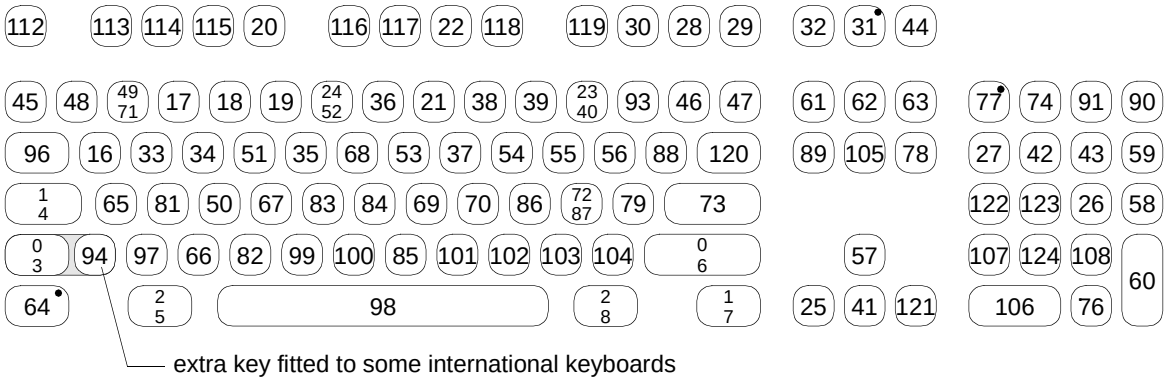


Figure 23.3 Internal key numbers for standard Archimedes keyboard

Keyboard used in portable machines

The keyboard used in portable machines generates these internal key numbers:

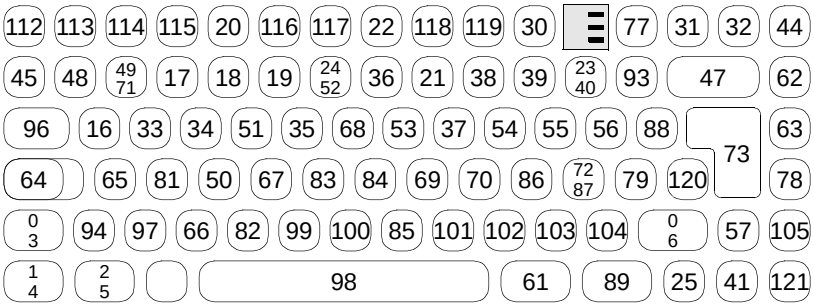


Figure 23.4 Internal key codes for keyboard used in portable machines

When the key to the left of the space bar is pressed (labelled ‘FN’ for UK machines), some keys generate different internal key codes, as shown below:

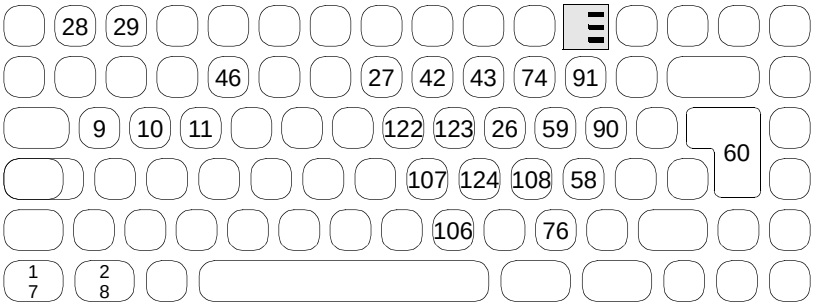


Figure 23.5 Alternative internal key codes for keyboard used in portable machines

You should not rely on the behaviour of other keys when pressed in conjunction with the FN key, as this may change in the future.

Service Calls

Service_KeyHandler (Service Call &44)

Keyboard handler

On entry

R1 = &44 (reason code)
R2 = keyboard ID

On exit

R1 preserved to pass on (don’t claim)
R2 preserved

Use

This call is made on reset, when the OS has established which type of keyboard is present, and after an OS_InstallKeyHandler SWI (see page 1-945). It is for the information of keyboard handler modules which need to know what sort of keyboard is present; it should not be claimed.

Standard Archimedes keyboards all have a keyboard ID of 1. The A4 internal keyboard, or a PC external keyboard, give a keyboard ID of 2, except under RISC OS 2, which does not support them.

SWI Calls

OS_ReadC (SWI &04)

Read a character from the input stream

On entry

—

On exit

if C flag = 0 then R0 = ASCII code
if C flag = 1 then R0 = error type: &1B in R0 means an escape

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 will read a character from the input stream. OS_Byte 2 can be used to change the selection of the current input stream.

Cursor key presses go into the buffer. When OS_ReadC reads a cursor key code from the buffer it handles the cursor editing for you, assuming the cursor keys are set up to do cursor editing. That is, if one of the arrow keys is pressed, cursor edit mode is entered, indicated by the presence of two cursors on the screen. You can copy characters from underneath the input cursor by pressing Copy. The character read is returned from the routine as if you had typed it explicitly.

Cursor editing only applies if enabled (see OS_Byte 4 on page 1-884), and is cancelled when ASCII 13 is sent to the VDU driver.

Related SWIs

OS_Byte 2 (page 1-882), OS_ReadLine (page 1-941)

Related vectors

RdchV

OS_Byte 2 (SWI &06)

Specify input stream

On entry

R0 = 2
R1 = stream selection (0, 1 or 2)

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 device from which all subsequent input is taken by OS_ReadC. This is determined by the value of R1 passed as follows:

- 0 for keyboard input with serial input buffer disabled
- 1 for serial input
- 2 for keyboard input with serial input buffer enabled

The difference between the 0 and 2 values is that the latter allows characters to be received into the serial input buffer under interrupts at the same time as the keyboard is being used as the primary input. If the input stream is subsequently switched to the serial device, then those characters can then be read.

Note that the value returned in R1 from this call is:

- 0 when input was from the keyboard
- 1 when input was from the serial port

The state of this variable can be read by OS_Byte 177.

Related SWIs

OS_Byte 177 (page 1-902)

Related vectors

ByteV

OS_Byte 4 (SWI &06)

Write cursor key status

On entry

R0 = 4
R1 = new state

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

This call alters the effect of the four arrow keys and the Copy key. The value of R1 determines their state:

0 Enables cursor editing. This is the default state.

1 Disables cursor editing. When pressed, the keys return the following ASCII values:

Key	Value
Copy	135
Left arrow	136
Right arrow	137
Down arrow	138
Up arrow	139

2 Cursor keys act as function keys. The function key numbers assigned are:

Key	Function key number
Copy	11
Left arrow	12
Right arrow	13
Down arrow	14
Up arrow	15

OS_Byte 237 may be used to write and read this state.

Related SWIs

OS_Byte 237 (page 1-929)

Related vectors

ByteV

OS_Byte 11 (SWI &06)

Write keyboard auto-repeat delay

On entry

R0 = 11
R1 = delay period in centiseconds

On exit

R0 preserved
R1 = previous delay period
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 must hold down each key on the keyboard for a number of centiseconds before it begins to autorepeat. This call enables you to change the initial delay from the default set by *Configure Delay.

If the delay period is zero, then auto-repeat is disabled.

This variable may also be read and set using OS_Byte 196.

Related SWIs

OS_Byte 12 (page 1-888), OS_Byte 196 (page 1-904), OS_Byte 197 (page 1-906)

Related vectors

ByteV

OS_Byte 12 (SWI &06)

Write keyboard auto-repeat rate

On entry

R0 = 12
R1 = repeat rate in centiseconds (unless R1 = 0)

On exit

R0 preserved
R1 = previous repeat rate
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

After the auto-repeat delay specified by OS_Byte 11, each key will repeat until released at the rate passed to this call. This call enables you to change the initial rate from the default set by *Configure Repeat. One particular use of this is to speed up cursor editing.

If the rate is zero, then the auto-repeat and delay values are reset to their configured settings.

This variable may also be read and set using OS_Byte 197.

Related SWIs

OS_Byte 11 (page 1-886), OS_Byte 196 (page 1-904), OS_Byte 197 (page 1-906)

Related vectors

ByteV

OS_Byte 18 (SWI &06)

Reset function key definitions

On entry

R0 = 18

On exit

R0 preserved
R1, 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 removes the system variables Key$0 to Key$15, which contain the function key definitions. It also cancels any key string currently being read.

You can also clear individual strings by *Key *n*, or all of them by *Unset Key$*. Neither of these commands cancel the current key expansion, though.

Related SWIs

None

Related vectors

ByteV

OS_Byte 118 (SWI &06)

Reflect keyboard status in LEDs

On entry

R0 = 118

On exit

R0 preserved
R1, 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 settings of Caps Lock, Scroll Lock and Num Lock are held in a location referred to as the keyboard status byte. See OS_Byte 202 on page 1-914 for detail of this.

Under normal circumstances they are shown by the keyboard LEDs which are set into the keycaps. However, if the keyboard status byte is written to using OS_Byte 202, then the LEDs will not update. This call ensures that the current contents of the keyboard status byte are reflected in the LEDs.

Related SWIs

OS_Byte 202 (page 1-914)

Related vectors

ByteV

OS_Byte 121 (SWI &06)

Keyboard scan

On entry

R0 = 121
R1 = key(s) to be detected

On exit

R0 preserved
R1 = if/which key has been detected
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 allows checking the keyboard to see whether a particular key or a range of keys is being pressed. It uses the internal key number (see the section entitled *Internal key numbers* on page 1-877 for a complete list).

Single key

To check for a single key, R1 must contain the internal key number exclusive ORd with &80 (R1 EOR &80). The value returned in R1 will be &FF if that key is currently down and zero if it is not.

Key range

To check for a range of key values, it is possible to set the ‘low tide’ mark. That is, no internal key number below the value in R1 on entry will be recognised. Since Shift, Ctrl, Alt and the mouse keys are at the bottom then this is very convenient.

The value returned in R1 will be the internal key number if a key is currently down or &FF if no key is down.

Related SWIs

OS_Byte 122 (page 1-894), OS_Byte 129 (page 1-899)

Related vectors

ByteV

OS_Byte 122 (SWI &06)

Keyboard scan (other than Shift, Ctrl, Alt and mouse keys)

On entry

R0 = 122

On exit

R0 preserved
R1 = internal key number of key, or &FF if none
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 allows checking the keyboard to see whether any key is being pressed. It uses the internal key number (see the section entitled *Internal key numbers* on page 1-877 for a complete list). All key numbers below 16 are ignored. This excludes all Shift, Ctrl, Alt and mouse keys. It is equivalent to calling OS_Byte 121 with R1 = 16.

Related SWIs

OS_Byte 121 (page 1-892), OS_Byte 129 (page 1-899)

Related vectors

ByteV

OS_Byte 124 (SWI &06)

Clear escape condition

On entry

R0 = 124

On exit

R0 preserved
R1, 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 clears any escape condition by calling the escape handler with R11 = 0, and then returns.

Related SWIs

OS_Byte 125 (page 1-896), OS_Byte 126 (page 1-897)

Related vectors

ByteV

OS_Byte 125 (SWI &06)

Set escape condition

On entry

R0 = 125

On exit

R0 preserved
R1, 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 is used to set the escape flag and call the escape handler. An escape event is not generated.

Related SWIs

OS_Byte 124 (page 1-895), OS_Byte 126 (page 1-897)

Related vectors

ByteV

OS_Byte 126 (SWI &06)

Acknowledge escape condition

On entry

R0 = 126

On exit

R0 preserved
R1 = 255 if escape condition has been cleared, or 0 if none to clear
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 attempts to clear an escape condition if one exists. It may or may not need to perform various actions to tidy up after the escape condition depending on whether the escape condition side effects (see OS_Byte 230) have been enabled or not.

The escape handler is called to indicate clearing of the escape condition.

The value returned in R1 indicates whether or not the escape condition has been cleared. &FF indicates success, while zero means that there was no escape condition to clear.

Related SWIs

OS_Byte 124 (page 1-895), OS_Byte 125 (page 1-896), OS_Byte 230 (page 1-927)

Related vectors

ByteV

OS_Byte 129 (SWI &06)

Read keyboard for information

On entry

R0 = 129

To read a key within a time limit:
R1 = time limit low byte
R2 = time limit high byte (in range &00 - &7F)

To read the OS version identifier:
R1 = 0
R2 = &FF

To scan the keyboard for a range of keys:
R1 = lowest internal key number EOR &7F (ie a value of &01 - &7F)
R2 = &FF

To scan the keyboard for a particular key:
R1 = internal key number EOR &FF (ie a value of &80 - &FF)
R2 = &FF

On exit

R0 preserved

If reading a key within a time limit:
R1 = ASCII code if character read, else undefined
R2 = &00 if character read, &1B if an escape condition exists, or &FF if timeout

If reading the OS version identifier:
R1 = &A0 for Arthur 1.20, &A1 for RISC OS 2.00, &A2 for RISC OS 2.01, &A3 for RISC OS 3 (version 3.00), or &A4 for RISC OS 3 (versions 3.10 and 3.11)
R2 = &00

If scanning the keyboard for a range of keys:
R1 = internal key number, or &FF if none pressed
R2 is corrupted

If scanning the keyboard for a particular key:
R1 = &FF if the required key was pressed, 0 otherwise
R2 = &FF if the required key was pressed, 0 otherwise

Interrupts

Interrupt status:

- Enabled when reading a key within a time limit
- Not altered for remaining three operations

Fast interrupts are enabled for all operations

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This OS_Byte is four separate operations in one:

- read an ASCII key value read from the keyboard with a timeout
- read the OS version identifier
- scan the keyboard for a range of keys
- scan the keyboard for a particular key.

Read key with time limit

In this operation, RISC OS waits up to a specified time for a key to be pressed, if there are none in the keyboard buffer.

The time limit is set according to the following calculation:

R1+(R2×256) centiseconds

The upper limit is 32767 centiseconds. To indicate the time of (n) centiseconds, then:

R1 = n MOD &100
R2 = n DIV &100

If an escape condition is detected during this operation it should be acknowledged by the application using OS_Byte 126, or cleared using OS_Byte 124.

While RISC OS is waiting for a keyboard character during one of these calls, it also deals with cursor key presses. That is, if one of the arrow keys is pressed, cursor edit mode is entered, indicated by the presence of two cursors on the screen. You can copy characters from underneath the input cursor by pressing Copy. The character read is returned from the routine as if you had typed it explicitly. Cursor editing is cancelled when Return (ASCII 13) is sent to the VDU driver. Cursor editing can be disabled with OS_Byte 4.

Read the OS version identifier

If R2=&FF and R1=0, then the OS version identifier is read.

Scan for a range of characters

If R2=&FF and R1 is in the range &1 to &7F, then the keyboard is scanned for any keys that are being pressed, which have an internal key number greater than or equal to R1 EOR &7F. If found, the internal key number is returned. If no key is found, then &FF is returned.

Scan for a particular key

If R2=&FF and R1 is in the range &80 to &7F, then the keyboard is scanned for a particular key with internal key number equal to R1 EOR &FF.

In BBC/Master series computers, the internal key numbers are the same as the keyboard scan numbers; but the two differ for other Acorn computers.

A list of all internal key numbers can be found in the section entitled *Internal key numbers* on page 1-877.

Related SWIs

OS_Byte 121 (page 1-892), OS_Byte 122 (page 1-894)

Related vectors

ByteV

OS_Byte 177 (SWI &06)

Read input stream selection

On entry

R0 = 177
R1 = 0
R2 = 255

On exit

R0 preserved
R1 = value of stream selection
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 returns the number of the buffer from which character input gets characters:

- 0 when input was from the keyboard
- 1 when input was from the serial port

You must not alter this number with this call by using other values in R1 and R2.

Related SWIs

OS_Byte 2 (page 1-882)

Related vectors

ByteV

OS_Byte 196 (SWI &06)

Read/write keyboard auto-repeat delay

On entry

R0 = 196
R1 = 0 to read, or new delay to write
R2 = 255 to read, or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 = keyboard auto-repeat rate (see OS_Byte 197)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The delay stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((delay AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call can read and set the keyboard auto-repeat delay value. OS_Byte 11 can also write this variable, and has more information about it.

Related SWIs

OS_Byte 11 (page 1-886), OS_Byte 12 (page 1-888), OS_Byte 197 (page 1-906)

Related vectors

ByteV

OS_Byte 197 (SWI &06)

Read/write keyboard auto-repeat rate

On entry

R0 = 197
R1 = 0 to read, or new rate to write
R2 = 255 to read, or 1 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 rate stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((rate AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call can read and set the keyboard auto-repeat rate value. OS_Byte 12 can also write this variable, and has more information about it. Note the difference between *FX 12,0 (which sets the auto-repeat rate and delay to their configured values) and *FX 197,0 (which sets the auto-repeat rate to zero).

Related SWIs

OS_Byte 11 (page 1-886), OS_Byte 12 (page 1-888), OS_Byte 196 (page 1-904)

Related vectors

ByteV

OS_Byte 198 (SWI &06)

Read/write *Exec file handle

On entry

R0 = 198
R1 = 0 to read, or new handle 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 handle stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((handle AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This command can be used to read or write the location that holds the Exec file handle.

If reading, it can tell whether an Exec file is the current input stream or not. Any non-zero number is a handle and hence the input stream.

If writing a handle over a zero, then it causes the same effect as a *Exec command.

If writing over a Exec file handle, the current Exec file will be switched off. This handle, which is returned, should then be properly closed after use. If you write a new handle value in its place, then this has the effect of switching input in mid-stream. If you write a zero in this case, then it will have terminate the current input stream.

In both these cases care must be taken not to cause the Exec file to stop at an inconvenient point.

If you are writing a file handle, the new file must be open for input or update, otherwise a Channel error occurs. If an attempt is made to use a write-only file for the *Exec file, a ‘Not open for reading’ error is given.

Related SWIs

None

Related vectors

ByteV

OS_Byte 200 (SWI &06)

Read/write Break and Escape effect

On entry

R0 = 200
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 = keyboard disable flag (see OS_Byte 201)

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) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call can read or change the effects of a reset (including resets caused by Break) and of Escape.

The bottom two bits of R1 have the following significance:

Bit	Value	Effect
0	0	Normal escape action
	1	Escape disabled unless caused by OS_Byte 125
1	0	Normal reset action
	1	Power on reset (only if bits 2 - 7 of R1 are zero) This means a value of 2_0000001x causes a memory clear.

Related SWIs

None

Related vectors

ByteV

OS_Byte 201 (SWI &06)

Read/write keyboard disable flag

On entry

R0 = 201
R1 = 0 to read, or new flag to write
R2 = 255 to read, or 0 to write

On exit

R0 preserved
R1 = 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

The flag stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((flag AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call allows you to read and change the keyboard state (ie whether the keyboard is enabled or disabled). When it is enabled, all keys are read as normal. When it is disabled, the keyboard interrupt service routine does not place these keys into the keyboard buffer.

A value of zero will enable keyboard input, while any non-zero value will disable it.

Related SWIs

None

Related vectors

ByteV

OS_Byte 202 (SWI &06)

Read/write keyboard status byte

On entry

R0 = 202
R1 = 0 to read, or new status to write (with bit 0 clear)
R2 = 255 to read, or 1 to write

On exit

R0 preserved
R1 = status before being overwritten
R2 = serial input buffer space (see OS_Byte 203)

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) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

The keyboard status byte holds information on the current status of the keyboard, such as the setting of Caps Lock. This call enables you to read and change these settings.

The bit pattern in R1 determines the settings. In this table, the State column has on and off in it. *On* means a LED is lit or a key is pressed, and *off* means the opposite. Take careful note of the state, because they are not all in the same order:

Bit	Value	State	Meaning
0	—	—	Reserved for use by keyboard handler: must be preserved when writing
1	0	off	Scroll Lock
	1	on	
2	0	on	Num Lock
	1	off	
3	0	off	Shift
	1	on	
4	0	on	Caps Lock
	1	off	
5			Normally set
6	0	off	Ctrl
	1	on	
7	0	off	Shift Enable
	1	on	

If Caps Lock is on, then Shift will have no effect on letters. If Shift Enable and Caps Lock are on, then Shift will get lower case. You can enter this state from the keyboard by holding Shift down and pressing Caps Lock.

This call does not update the LEDs. The next key down or up event will update them, or you can call OS_Byte 118.

Related SWIs

OS_Byte 118 (page 1-891)

Related vectors

ByteV

OS_Byte 216 (SWI &06)

Read/write length of function key string

On entry

R0 = 216
R1 = 0 to read, or new length to write
R2 = 255 to read, or 0 to write

On exit

R0 preserved
R1 = length before being overwritten
R2 = paged mode line count (see OS_Byte 217)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The length stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((length AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call reads and changes the count of characters left in the currently active function key definition. An active function key is one that is being read by OS_ReadC instead of the current input stream.

If the length is zero, then no function key string is being read. A zero length must never be changed with this call.

A non-zero value shows that a function key string is active. Setting it to zero effectively cancels that function key from that point. Changing it to any non-zero value will have an indeterminate effect.

Related SWIs

OS_ReadC (page 1-880)

Related vectors

ByteV

OS_Byte 219 (SWI &06)

Read/write Tab key value

On entry

R0 = 219
R1 = 0 to read, or new 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) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

OS_Byte 219 reads or modifies the code inserted into the keyboard buffer when the Tab key is pressed (the default is 9). If the value specified is in the range &80 to &FF, then the value to be inserted is modified by the state of the Shift and Ctrl keys as follows:

- Shift exclusive ORs the value with &10
- Ctrl exclusive ORs the value with &20

The value inserted will be interpreted by OS_ReadC in the normal way. For example, if the value specified is &82, then the Tab key behaves in an identical way to the function key F2.

Related SWIs

OS_ReadC (page 1-880)

Related vectors

ByteV

OS_Byte 220 (SWI &06)

Read/write escape character

On entry

R0 = 220
R1 = 0 to read, or new 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) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call can read and change the character that will cause an escape condition when it is read from the input stream. Escape (ASCII 27) is the default.

For example:

Value	Key that causes an escape condition
27	Escape
53	‘5’
&81	F1
&A1	Ctrl F1

Related SWIs

OS_ReadC (page 1-880)

Related vectors

ByteV

OS_Bytes 221 - 228 (SWI &06)

Read/write interpretation of buffer codes

On entry

R0 = 221 - 228
R1 = 0 to read, or new 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) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call provides a way of reading and changing how the codes from &80 to &FF are interpreted when read from the input buffer.

They are split into eight groups as follows:

OS_Byte	Range of buffer codes controlled
221	&C0 - &CF
222	&D0 - &DF
223	&E0 - &EF
224	&F0 - &FF
225	&80 - &8F
226	&90 - &9F
227	&A0 - &AF
228	&B0 - &BF

The list below shows the keys that can produce codes in these groups:

Key	Code	+Shift	+Ctrl	+Ctrl-Shift
Print	&80	&90	&A0	&B0
F1	&81	&91	&A1	&B1
F2	&82	&92	&A2	&B2
:	:	:	:	:
F9	&89	&99	&A9	&B9
Copy	&8B	&9B	&AB	&BB
←	&8C	&9C	&AC	&BC
→	&8D	&9D	&AD	&BD
↓	&8E	&9E	&AE	&BE
↑	&8F	&9F	&AF	&BF
Page Down	&9E	&8E	&BE	&AE
Page Up	&9F	&8F	&BF	&AF
F10	&CA	&DA	&EA	&FA
F11	&CB	&DB	&EB	&FB
F12	&CC	&DC	&EC	&FC
Insert	&CD	&DD	&ED	&FD

These SWIs only affect the codes generated by the Copy and arrow keys if they have been set up to act as function keys by calling OS_Byte 4 with R1 = 2. Normally this is not the case, and you should use OS_Byte 4 to control the action of these keys.

Also, when a reset occurs, the code &CA is inserted into the input buffer. This causes the key definition for function key 10 to be used for subsequent input if it is defined.

Some of these codes cannot be generated from the main keyboard, but must be produced via one of the following techniques:

- use these calls to generate them with keys
- re-base the numeric keypad with OS_Byte 238
- insert into the buffer with OS_Byte 138
- insert into the buffer with OS_Byte 153
- receive via the serial input port

The interpretation of these codes depends upon the value of R1 passed. This is the interpretation value. It determines what action will be taken with a code in the appropriate block:

Value	Interpretation
0	discard the code
1	generates the string assigned to function key (code MOD 16)
2	generates a NULL (ASCII 0) followed by the code
3 - &FF	acts as offset: ie (code MOD 16) + value

If any block has been set to interpretation value 2, then a Ctrl-@ (ASCII 0) will be passed as two zeros to differentiate it from a high code. This mode is used with software that can cope with the international character set in the range &A0 - &FF. It is recommended that the function keys return a NULL followed by the key code, so that they can be distinguished from actual ASCII characters in this range.

This is the default setting for each of the blocks:

Block	Default	Interpretation
&80 - &8F	1	function keys
&90 - &9F	&80	return (buffer code – &10)
&A0 - &AF	&90	return (buffer code – &10)
&B0 - &BF	0	discard
&C0 - &CF	1	function keys
&D0 - &DF	&D0	return buffer code unchanged
&E0 - &EF	&E0	return buffer code unchanged
&F0 - &FF	&F0	return buffer code unchanged

Related SWIs

OS_Byte 4 (page 1-884), OS_Byte 138 (page 1-172), OS_Byte 153 (page 1-178), OS_Byte 238 (page 1-931)

Related vectors

ByteV

OS_Byte 229
(SWI &06)

Read/write Escape key status

On entry

R0 = 229
R1 = 0 to read, or new status to write
R2 = 255 to read, or 0 to write

On exit

R0 preserved
R1 = status before being overwritten
R2 = escape effects (see OS_Byte 230)

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) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call allows you to enable or disable the generation of escape conditions, and to read the current setting. Escape conditions may be caused by pressing the current escape character or by inserting it into the input buffer with OS_Byte 153.

If the value of R1 passed is zero, which is the default, then escape conditions are enabled. Any non-zero value will disable them. When they are disabled, the current escape character set by OS_Byte 220 will pass through the input stream unaltered.

OS_Byte 200 can also control the enabling of escape conditions.

Related SWIs

OS_Byte 153 (page 1-178), OS_Byte 200 (page 1-910), OS_Byte 220 (page 1-920)

Related vectors

ByteV

OS_Byte 230 (SWI &06)

Read/write escape effects

On entry

R0 = 230
R1 = 0 to read, or new status to write
R2 = 255 to read, or 0 to write

On exit

R0 preserved
R1 = status 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 status stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((status AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

By default, the acknowledgement of an escape condition produces the following effects:

- Flushes all active buffers
- Closes any currently open *Exec file
- Clears the VDU queue
- Clears the VDU line count used in paged mode
- Terminates the sound being produced.

This call enables you to determine whether the escape effects are currently enabled or disabled, and to change the setting if required.

If the value of R1 passed is zero, which is the default, then escape effects are enabled. Any non-zero value will disable them.

Related SWIs

None

Related vectors

ByteV

OS_Byte 237 (SWI &06)

Read/write cursor key status

On entry

R0 = 237
R1 = 0 to read, or new state to write
R2 = 255 to read, or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 = numeric keypad interpretation (see OS_Byte 238)

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) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This can read and modify the cursor key status. OS_Byte 4 can perform an identical write operation. See the description of that SWI in this chapter for details of the status.

Related SWIs

OS_Byte 4 (page 1-884)

Related vectors

ByteV

OS_Byte 238 (SWI &06)

Read/write numeric keypad interpretation

On entry

R0 = 238
R1 = 0 to read, or new 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) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call controls the character which is inserted into the input buffer when you press one of the keypad keys. The inserted character is derived from the sum of a base value (set by this call) and an offset, which depends on the key pressed. The inner (lighter) keys have two different offsets. The offset used depends on the state of Num Lock.

By default, the base number is 48: ie they generate codes which are displacements from 48 (ASCII ‘0’).

This table shows the effect of the default settings on the keypad:

Key	Base Offset	Character Generated	Num Lock Offset	Character Generated
0	0	0	+157	Insert
1	+1	1	+91	Copy
2	+2	2	+94	Down
3	+3	3	+110	Page Down
4	+4	4	+92	Left
5	+5	5	ignored	
6	+6	6	+93	Right
7	+7	7	–18	Home
8	+8	8	+95	Up
9	+9	9	+111	Page Up
.	–2	.	+79	Delete
/	–1	/	unchanged	
*	–6	*	unchanged	
#	–13	#	unchanged	
–	–3	–	unchanged	
+	–5	+	unchanged	
Enter	–35	Return	unchanged	

Unlike the function keys, you can set the numeric keypad base number to any value in the range 0 - 255. (If a generated code lies outside this range it is reduced MOD 256). If a character generated by the numeric keypad is in the range &80 to &8F, then it will act like a soft function key.

OS_Byte 254 controls how Shift and Ctrl act upon numeric keypad characters.

Related SWIs

OS_Byte 254 (page 1-937)

Related vectors

ByteV

OS_Byte 247
(SWI &06)

Read/write Break key actions

On entry

R0 = 247
R1 = 0 to read, or new 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) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call reads and changes the result of pressing Break. The value byte alters Break and modifiers of it as follows:

Bits	Key Combination
0,1	Break
2,3	Shift Break
4,5	Ctrl Break
6,7	Ctrl Shift Break

Each two bit number may take on one of these values:

Value	Effect
00	Act as Reset
01	Act as escape key
10	No effect
11	Undefined

The default is 2_00000001, so Break causes an escape condition, and Shift-Break, Ctrl-Break and Ctrl-Shift-Break all act as resets.

Related SWIs

None

Related vectors

ByteV

OS_Byte 253 (SWI &06)

Read last reset type

On entry

R0 = 253
R1 = 0
R2 = 255

On exit

R0 preserved
R1 = break type
R2 = effect of Shift on keypad (see OS_Byte 254)

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 type of the last reset performed in R1:

Value	Reset type
0	Soft reset
1	Power-on reset
2	Hard reset

Related SWIs

None

Related vectors

ByteV

OS_Byte 254 (SWI &06)

Read/write effect of Shift and Ctrl on numeric keypad

On entry

R0 = 254
R1 = 0 to read, or new 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) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call allows you to enable or disable the effect of Shift and Ctrl on the numeric keypad or to read the current state. These keys may modify the code just before it is inserted into the input buffer.

If the value of R1 passed is zero, then Shift and Ctrl are enabled. Any non-zero value will disable them; this is the default.

If they are enabled then the following actions occur depending on the value generated by a key:

- if the value ≥ &80:
  Shift exclusive ORs the value with &10
  Ctrl exclusive ORs the value with &20
- if the value < &80:
  Shift and Ctrl still have no effect

Related SWIs

None

Related vectors

ByteV

OS_Word 0 (SWI &07)

Read a line from input stream to memory

On entry

R0 = 0
R1 = pointer to parameter block

On exit

R0 preserved
R1 = preserved (and parameter block unaltered)
R2 = length of input line, not including the Return
the C flag is set if input is terminated by an escape condition

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 equivalent to OS_ReadLine, but has restrictions on the location of the character buffer used. It is provided for compatibility with older Acorn operating systems.

The parameter block pointed to has the following structure:

Offset	Purpose	Equivalent in OS_ReadLine
0	LSB of buffer address	R0
1	MSB of buffer address	
2	size of buffer	R1
3	lowest ASCII code	R2
4	highest ASCII code	R3

Because the parameter block only uses 2 bytes to specify the character buffer's address, it must lie in the bottom 64K of memory. Furthermore, the range &0000 to &7FFF is reserved for RISC OS, so in fact the buffer must lie in the range &8000 to &FFFF.

Related SWIs

OS_ReadLine (page 1-941)

Related vectors

WordV

OS_ReadLine (SWI &0E)

Read a line from the input stream

On entry

R0 = pointer to buffer to hold the line (bits 0-29), and flags (bits 30-31)
bit 31 set ⇒ echo only those characters that enter the buffer
bit 30 set ⇒ echo characters by echoing the character in R4
R1 = size of buffer
R2 = lowest ASCII value to pass
R3 = highest ASCII value to pass
R4 = character to echo if bit 30 of R0 is set

On exit

R0 corrupted
R1 = length of buffer read, not including Return.
R2, R3 corrupted
the C flag is set if input is terminated by an escape condition

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_ReadLine reads a line of text from the current input stream using OS_ReadC. Characters in the range specified by R2 and R3 are placed in a read buffer, the address of which is given in R0. These characters are also echoed to OS_WriteC; if bit 31 of R0 is clear on entry, characters outside the range are also echoed. Alternatively – by setting bit

30 of R0 – you can echo the character held in R4, rather than the actual character that was read. This is useful, for example, to read a password without echoing its actual characters to the screen.

Certain characters and conditions are specially treated:

- A carriage return (ASCII 13) or a linefeed terminates input. A carriage return is placed in the read buffer, but the length returned in R1 will not include it. A carriage return and a linefeed are echoed to OS_WriteC.
- An escape condition also terminates input. This can represent the escape key being pressed, but it can also be caused by other means, such as an OS_Byte 125.
- A delete (ASCII 127) or a backspace (ASCII 8) character act in the same way. If there are no characters in the read buffer they have no effect. Otherwise they each remove the character last written into the buffer, and echo a delete character to OS_WriteC.
- Ctrl-U (ASCII 21) acts similarly to delete. Again, if there are no characters in the read buffer it has no effect. Otherwise it removes all the characters in the buffer, and echoes that many delete characters to OS_WriteC, effectively erasing the line.

If the number of characters input reaches the number passed in R1, further characters are ignored and cause Ctrl-G (ASCII 7) to be sent to OS_WriteC, which will normally cause a sound to be emitted. The deleting keys mentioned above will still function.

You must not call OS_ReadLine from an interrupt or event routine.

Related SWIs

OS_WriteC (page 1-515), OS_ReadC (page 1-880), OS_Word 0 (page 1-939)

Related vectors

ReadLineV, WrchV

OS_ReadEscapeState (SWI &2C)

Check whether an escape condition has occurred

On entry

—

On exit

the C flag is set if an escape condition has occurred

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadEscapeState sets or clears the carry flag depending on whether escape is set or not. Once an escape condition has been detected (either through this call or, for example, with OS_ReadC), it should be acknowledged using OS_Byte 126 or cleared using OS_Byte 124.

This call is useful if a program is executing in a loop which the user may want to escape from, but isn't performing any input operations which would let it know about the escape.

Note that OS_ReadEscapeState may be called from an interrupt routine. However, OS_Byte 126 may not be, so if an escape is detected under interrupts, the interrupt routine must set a flag which is checked by the foreground task, rather than attempt to acknowledge the escape itself.

Related SWIs

OS_Byte 124 (page 1-895), OS_Byte 126 (page 1-897)

Related vectors

None

OS_InstallKeyHandler (SWI &3E)

Install a key handler or read the address of the current one

On entry

R0 = 0 to read address of current keyboard handler
1 to read keyboard ID from keyboard
>1 to set address of new keyboard handler

On exit

R0 = address of current/old keyboard handler, or keyboard ID

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_InstallKeyHandler installs a new keyboard handler to replace the default code. Alternatively you can read the address of the current handler, or read the keyboard ID. The returned keyboard ID may be:

Value	Meaning
1	standard Archimedes keyboard
2	A4 internal keyboard, or PC external keyboard

Related SWIs

None

Related vectors

None

*Commands

*Configure Caps

Sets the configured value for Caps Lock to ON

Syntax

```
*Configure Caps
```

Parameters

None

Use

*Configure Caps sets the configured value for Caps Lock to ON, so that when you switch on or reset your machine, you will start typing in capital letters. This was the default value on RISC OS 2; NoCaps is the default value from RISC OS 3 onwards.

Example

```
*Configure Caps
```

Related commands

*Configure NoCaps, *Configure ShCaps

Related SWIs

OS_Byte 202 (page 1-914)

Related vectors

None

*Configure Delay

Sets the configured delay before keys start to auto-repeat

Syntax

`*Configure Delay` *n*

Parameters

n delay (in centiseconds)

Use

*Configure Delay sets the configured delay before keys start to auto-repeat. A value of zero disables auto-repeat. The default value is 32.

Example

```
*Configure Delay 20
```

Related commands

*Configure Repeat

Related SWIs

OS_Byte 11 (page 1-886)

Related vectors

None

*Configure NoCaps

Sets the configured value for Caps Lock to OFF

Syntax

```
*Configure NoCaps
```

Parameters

None

Use

*Configure NoCaps sets the configured value for Caps Lock to OFF, so that when you switch on or reset your machine, you will start typing in lower case. This is the default value from RISC OS 3 onwards; Caps was the default value on RISC OS 2.

Example

```
*Configure NoCaps
```

Related commands

*Configure Caps, *Configure ShCaps

Related SWIs

OS_Byte 202 (page 1-914)

Related vectors

None

*Configure Repeat

Sets the configured interval between the generation of auto-repeat keys

Syntax

```
*Configure Repeat n
```

Parameters

n interval (in centiseconds)

Use

*Configure Repeat sets the configured interval between the generation of auto-repeat keys. A value of zero sets an infinite interval, so the character repeats just once, after the auto-repeat delay. To completely disable auto-repeat, set the delay to zero; *Configure Delay 0 will do this.

The default value is 8.

Example

```
*Configure Repeat 3
```

Related commands

*Configure Delay

Related SWIs

OS_Byte 12 (page 1-888)

Related vectors

None

*Configure ShCaps

Sets the configured value for Caps Lock to ON, Shift producing lower case letters

Syntax

```
*Configure ShCaps
```

Parameters

None

Use

*Configure ShCaps sets the configured value for Caps Lock to ON, so that when you switch on or reset your machine, you will start typing in capital letters. Holding down the Shift key will produce lower case letters, which does not happen when Caps is the configured value. Caps is the default value from RISC OS 3 onwards; Caps was the default value on RISC OS 2.

Example

```
*Configure ShCaps
```

Related commands

*Configure NoCaps, *Configure Caps

Related SWIs

OS_Byte 202 (page 1-914)

Related vectors

None

*Key

Assigns a string to a function key

Syntax

`*Key` *`keynumber`* [*`string`*]

Parameters

`keynumber`	a number from 0 to 15
`string`	any GSTrans-compatible string

Use

*Key assigns a string to a function key. It provides a very simple way of setting up function keys so that repetitive or error-prone strings (such as complex commands) can be initiated with a single keystroke. You can use any string up to 255 characters long.

The string is transformed by GSTrans before being stored. This means that you can, for example, represent Return using '|M' (as in the example below). See the section entitled *GS string operations* on page 1-454 for details.

The string is stored in the system variable `Key$`*`keynumber`*, for example Key$1 for function key 1. This enables a key's definition to be read before it is used, and manipulated like any other variable. Also, because a key string can be set as a macro, its value may be made to change each time it is used.

In addition to F1 to F12, these keys can act as function keys by default:

- Print as F0
- Insert as F13

and these keys can be made to act as function keys by the command *FX4,2:

- Copy as F11
- left arrow as F12
- right arrow as F13
- down arrow as F14
- up arrow as F15

Function keys are generally unaffected by a soft break, but lost following a hard break.

Example

```
*Key 8 *Audio On|M *Speaker On|m *Volume 127|m
*SetMacro Key$1 ||The time is <Sys$Time>|m
```

Related commands

*Set, *SetMacro

Related SWIs

OS_SetVarVal (page 1-316)

Related vectors

None

24 The CLI

Introduction

There are two ways in which you can interact with the OS and the various modules which provide extensions to it. The first way is to call one of the many SWI routines provided, such as OS_Byte, OS_ReadMonotonicTime, Wimp_Initialise etc. The SWI interface provides an efficient calling mechanism for use within programs in any language.

However, for users wishing to issue commands to the operating system, the SWI interface is not so convenient. As it is difficult to remember SWI names, reason codes, register contents on entry and exit, etc, the *command line interpreter* (CLI) interface is often used. Using this technique, you enter a textual command string, possibly followed by parameters, which is then passed by the application to the OS. The OS tries to decode the command and carry out the appropriate action. If the command is not recognised by the OS, the other modules in the system try to execute the command instead.

The CLI interface is a powerful one because the OS performs a certain amount of pre-processing on the line before it attempts to interpret it. For example, variable names may be substituted in the parameter part of the line, and command aliases may be used.

By convention, an application passes commands to the OS if they are prefixed by the * character. For example, from the BASIC '>' prompt, any OS command may be issued simply by making * the first non-space character on the line. The * is not part of the command; the OS, in fact, strips any leading *s and spaces from a command before it tries to decode it.

Some languages also provide built-in statements which can be used to perform an OS command. Again, BASIC provides the OSCLI statement, which evaluates a string expression and passes this to the OS command line interpreter. The 'C' language provides the system() function for the same purpose.

Overview and Technical Details

A program can call the CLI using the SWI OS_CLI (page 1-961). This simply passes a string from the program to the CLI to be interpreted. If you wish to allow the user to type a number of CLI commands, then you can pass 'GOS' (see page 1-965) as the string to OS_CLI. See the chapter entitled *Program Environment* on page 1-287, for information on how to set up RISC OS to return to your program when the user types *Quit.

CLI effects

When a CLI command is received by the kernel, it performs a number of operations upon it. Note that in most cases, the case of commands is ignored. Only if you are creating something with a name is the case kept. The sections below go through each of these.

Leading characters

Certain leading characters will be treated in a special way:

'*'	all leading stars are discarded
' '	all leading spaces are discarded
'|'	this indicates that the line is a comment, and will be ignored
'/'	treat the rest of the command as if it had been prefixed with *Run
'%'	skip alias checking.
'–'	override current filing system name: eg –adfs–
'.'	check for Alias$. and use *Cat if it doesn't exist

Apart from '%' and '–', the above commands should be self-explanatory. '%' is used to access a built-in command that currently has an alias overriding it; see the section entitled *Aliases*. For more information on '–', see the section below on *Context overriding*.

Context overriding

The currently selected filing system can be overridden in two different ways. The command can be prefixed with –name– or name:, where name is the name of a filing system or module. That is, you supply an absolute name of the filing system or module to send the command to. This gets around the problem of having to select the other filing system, perform the command and then re-enter the original filing system. For example, if you are on the net and want to look at a file on the current adfs device, the sequence of commands:

```
*adfs
*Info Fred
*net
```

can be replaced with either:

```
*–adfs–Info Fred
```

or even more succinctly:

```
*adfs:Info Fred
```

Here are some examples of overrides:

```
*–net–cat
*SpriteUtils:Slist
```

Note that if you are using –net– or net:, you cannot specify nodes on the net: eg –net#spqr–. This is because the command prefix only alters the filing system selected for the command. The part of an object specification after the '#' character is not part of the filing system name but is part of the object name. For example, if you wish to issue a command such as:

```
*net#oz:info fred
```

you can use instead:

```
*net:info #oz:fred
```

Redirection

Normally, input comes from the keyboard and output goes to the screen. Redirection allows this source and destination to be changed to any file or device. Output redirection can be viewed as having a *Spool file open for the duration of the command, and disabling all streams except for that one. Input redirection is like having a *Exec file open for the duration of the command.

The syntax of a *redirection specifier* is:

```
{ redirection_command [redirection_commands] }
```

where each *redirection_command* may be any of:

`> filename`	Output goes to filename
`< filename`	Input read from filename
`>> filename`	Output appended to filename

The redirection specifier can appear anywhere in a line. Note that there must be exactly one space between each element, or it will not be recognised as one. After being decoded. The redirection specifier is stripped before the rest of the command is interpreted. You can put as many redirection commands as you like within the curly brackets; however, only the last one in a given direction will be acted on.

Here are some examples of redirection:

```
*Cat { > mycat }
*Lex { > printer: }
*BASIC –quit { < answers } prog
*fred { < infile > outfile }
*Cat { > out1 < infile > out2 }
```

The fourth example shows how redirections can be concatenated within the same pair of braces.

In the final example, `out1` will be created with nothing in it, input will be read from `infile` and output will go to `out2`.

Abbreviations

Commands may be abbreviated by terminating them with a '.'. For example, you could type '*Mod.' instead of '*Modules'. When the CLI finds a terminating '.', it remembers that the command is an abbreviation, and when trying to match it to possible aliases or commands to execute (see below), it is satisfied if the abbreviation is a leading substring of the alias or command, rather than an exact match.

It is very dangerous to use abbreviations in programs, as they are dependent on the environment in which they are run. The command they execute can be changed completely by the presence of an alias, or by any addition of commands and change in module ordering under different releases of RISC OS.

Aliases

An alias is a variable of the form Alias$*cmd*, where *cmd* is the command name to match, made up from alphanumeric characters and these others:

! ' ( ) + – . ; = ? @ [ ] _ ` { } ~

If an alias exists which matches the current * Command, RISC OS obtains the value of the variable and replaces any of %0 to %9 in the value by the parameters, separated by spaces, that it reads on the rest of the input line. %*n in an alias stands for the rest of the command line, from parameter 'n' onwards.

Any unused parameters, which are given, are directly appended to the alias. The OS then recursively calls OS_CLI for all lines in the expanded value. However, it may give up at this stage if either the stack or its buffer space becomes full. For example, suppose this command is issued:

```
*SetPS 0.235
```

Suppose further that a variable exists called Alias$SetPS, and that this has the value –NET–PS %0|MConfigure PS %0. The OS will match the command name against the alias variable. It will then substitute all occurrences of %0 in the variable's value by 0.235. Then, the two lines of the variable will be executed thus:

```
–NET–PS 0.235
Configure PS 0.235
```

So, the net effect of executing the original command is to set the network printer server both temporarily, and also in the permanent configuration.

Another example using the parameter substitution is

```
*Set Alias$Mode Echo |<22>|<%0>
```

The '|'s before the angle brackets are to stop them from being evaluated when the *Set command is entered. Typing `*Mode n` will then set the display to mode 'n'.

Look-up the command

After all the previous steps have been completed, the command that is left after pre-processing must be executed. This is a list in order of the things that RISC OS will check to execute a command:

1. A check is made to see whether the command is in the kernel.
2. The kernel checks each module to see whether it supplies the command. Modules are checked in the order of the module list, as printed by the *Modules command.

   Amongst the modules checked is the filing system manager, File Switch, which contains those commands that apply to all filing systems, such as *Cat.
3. After the module search is complete, the kernel inspects the filing system specific commands in the current filing system module.
4. If the command is not recognised by the filing system module, the kernel issues an 'unknown command' service call.

   If the net is the current filing system, the command is sent to the file server, to see if the command is implemented there. For example, *Pass is implemented in this way.
5. If the command is still not recognised, then an attempt will be made to *Run it using the current path. The result of this *Run is passed back to the user.

Reading CLI parameters

If you are writing a module, the chances are that you will want to recognise one or more * Commands. The chapter entitled *Modules* on page 1-201 explains how you can cause the OS to recognise commands for you, and pass control to your module when one has been found. This section describes the OS calls which are available to facilitate the decoding of the rest of the command line.

The calls mentioned here may also be used by * Commands activated in other ways, eg a transient command loaded from disc. However, the way in which the tail of the command line is discovered will vary for these types of commands. See the chapter entitled *Program Environment* on page 1-287 for details.

On entry to your * Command routine, R0 contains a pointer to the ‘tail’ of the command, ie the first character after the command name itself (with spaces skipped). R1 contains the number of parameters, where a parameter is regarded as a sequence of characters separated by spaces.

The way in which the command uses the parameters depends on what it is doing. First, if there are too many or too few parameters, an error could be given. (A module can arrange for the OS to do this automatically.)

If a parameter is to be regarded as a string, OS_GSTrans may be used to decode any special sequences, eg control codes, variable names etc. If the parameter is a number, OS_ReadUnsigned might be used to convert it into binary. Finally, OS_EvaluateExpression could be used to read a whole arithmetic or string expression, and return the result in a buffer.

These calls are documented in the chapter entitled *Conversions* on page 1-453.

Note that the convention in RISC OS is to have parameters separated by spaces. Some of the built-in commands which have been carried over from the BBC/Master machines also allow commas. You should not support this option.

SWI Calls

OS_CLI (SWI &05)

Process a supervisor command

On entry

R0 = pointer to string terminated by Null, Linefeed or Return

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

OS_CLI will execute a string passed to it as if it had been typed in at the supervisor command line. When it is called, it performs the following actions:

Check stack space

The OS needs a certain amount of workspace to deal correctly with a command. If this is not available, the error ‘No room on supervisor stack’ will be generated.

Check command length

A * Command line must be less than or equal to 256 bytes long, including the terminating character. If it is not, the error ‘Too long’ is returned.

Execute command

The command is then passed to the command line interpreter and executed as any other * Command. This is described in the *Overview and Technical Details*.

Related SWIs

None

Related vectors

CLIV

OS_ChangeRedirection (SWI &5E)

Read or write OS_CLI input/output redirection handles

On entry

R0 = new file handle for input
    0 = not redirected
    –1 = leave alone
R1 = new file handle for output
    0 = not redirected
    –1 = leave alone

On exit

R0 = old file handle for input
    0 = not redirected
R1 = old file handle for output
    0 = not redirected

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI reads or writes the file handles used by OS_CLI to redirect input/output. It is mainly provided for the use of the TaskWindow module, but you may also find the call useful.

Related SWIs

None

Related vectors

None

* Commands

*GOS

Calls Command Line mode, and hence allows you to type * Commands

Syntax

```
*GOS
```

Parameters

None

Use

*GOS starts the RISC OS Supervisor application from the current environment. The supervisor can only execute *Commands.

This is useful for entering simple commands for immediate execution, or for testing longer sequences of commands – while building command line scripts –on a line-by-line basis.

However you should be careful when calling it from the middle of an application which does not ‘shell’ new applications. For example, calling *GOS in the middle of writing a BASIC program will mean that you will lose all of your unsaved work.

From the desktop, pressing F12 has a similar effect. To return to the desktop, press Return at the start of a line with the Supervisor prompt (‘*’). If you do not have this prompt, you will first have to type *Quit to leave the application you are using.

See the section entitled *Overview and Technical Details* on page 1-956 for a description of how the command line interface works.

Related commands

*Quit, *Desktop

Related SWIs

None

Related vectors

None

25 The rest of the kernel

Introduction

Kernel calls and commands are covered here that do not merit a chapter by themselves.

The following SWIs are described:

The following * Commands are also described:

SWI Calls

OS_Byte 0 (SWI &06)

Display OS version or return machine type

On entry

R0 = 0
R1 = 0 to display OS version string, or 1 to return machine type

On exit

R0 preserved
R1 = machine type if R1 = 1 on entry
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

If this is called with R1=0, then an error is returned, the text of which shows the version number and creation date of the operating system. If this is called with R1 = 1, then the machine type is returned in R1. All RISC OS computers return a machine type of 6; earlier non-RISC OS Acorn machines return different values.

Related SWIs

None

Related vectors

ByteV

OS_HeapSort (SWI &4F)

Heap sort a list of objects

On entry

R0 = number of elements to sort
R1 = pointer to array of word size objects, and flags in top 3 bits
R2 = type of object (0 - 5), or address of comparison routine
R3 = workspace pointer for comparison procedure (only needed if R2 > 5)
R4 = pointer to array of objects to be sorted (only needed if flag(s) set in R1)
R5 = size of an object in R4 (only needed if flag(s) set in R1)
R6 = address of temporary workspace of R5 bytes
(only needed if R5 > 16k or bit 29 of R1 is set)

On exit

R0 - R6 preserved

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 SWI will sort a list of any objects using the heap sort algorithm. Details of this algorithm can be found in:

Sorting and Searching D.E. Knuth (1973) Addison-Wesley, Reading Massachusetts, pages 145 -149.

It is not as fast as a quicksort for average sorts, but uses no extra memory beyond that which is initially passed in.

Basic usage

Used in the simplest way, only R0, R1 and R2 need be set up. R0 contains the number of objects that are in the list. R1 points to an array of word-sized entries. The value of R2 controls the interpretation of this array:

R2 value	Treat R1 as pointing to an array of…
0	cardinal (unsigned integer)
1	integer
2	pointer to cardinal
3	pointer to integer
4	pointer to characters (case insensitive)
5	pointer to characters (case sensitive)
>5	pointer to custom object In this last case, R2 is the address of the comparison routine

Comparison routine

If the R2 value is less than 6, then this call will handle sorting for you. If you want to sort any other kind of object, then you must provide a routine to compare two items and say which is the greater. Using this technique, any complex array of structures may be sorted. If you wish to use a comparison routine, then R2 contains the address of it. R3 must be set up with a value, usually a workspace pointer.

When called, the comparison routine is entered in SVC mode, with interrupts enabled. R0 and R1 contain two objects from the array passed to this SWI in R1. What they represent depends on what the object is, but in most cases they would be pointers to a structure of some kind. R12 contains the value originally passed in R3 to this SWI. Usually this is a workspace pointer, but it is up to you what it is used for.

Whilst in this routine, R0 - R3 may be corrupted, but all other registers must be preserved. The comparison routine returns a less than state in the flags if the object in R0 is less than the object in R1. A greater or equal state must be returned in the flags if the object in R0 is greater than or equal to the object in R1.

Advanced features

In cases where R2 is greater than 1, then there are two arrays in use. The word sized array of pointers pointed to by R1 and the ‘real’ object array. You can supply the address of this real array in R4 and the size of each object in it in R5. If this is done, then a number of optional actions can be performed. The top bits in R1 can be used as follows:

Bit	Meaning
29	use R6 as workspace
30	build word-array of pointers pointed to by R1 from R4,R5
31	sort true objects pointed to by R4 after sorting the pointers

Bit 30 is used to build the pointer array pointed to by R1 using R4 and R5 before sorting is started. It will create an array of pointers, where the first pointer points to the first object, the second pointer to the second object and so on. After sorting, these pointers will be jumbled so that the first pointer points to the ‘lowest’ object and so on.

Bit 31 is used to sort the real objects pointed to by R4 into the order described by the pointers in the array pointed to by R1 after sorting is complete. It may optionally be used in conjunction with bit 30.

If the size in R5 is greater than 16 Kbytes or if bit 29 is set in R1, then a pointer to workspace must be passed in R6. This points to a block R5 bytes in length. One reason for setting bit 29 is that this SWI will otherwise corrupt the RISC OS scratch space.

Related SWIs

None

Related vectors

None

OS_Confirm (SWI &59)

Get a yes or no answer

On entry

—

On exit

R0 = key that was pressed, in lowercase
the C flag is set if an escape condition occurred
the Z flag is set if the answer was Yes

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI gets a yes or no answer from the user. If the mouse pointer is visible, then it changes it to a three button mouse shape. The left button indicates yes, while the other two indicate no. On the keyboard, a key appropriate to the territory indicates yes, and any other key indicates no.

You should always check whether the answer was yes or no by testing the Z flag, rather than the value returned in R0; this ensures that your program will not need modifying for use with different territories.

The result in R0 is returned in lowercase, irrespective of the keyboard state. It is made available should you need to reflect a character to the screen.

An escape condition will abort the SWI and return with the C flag set.

Related SWIs

None

Related vectors

None

OS_CRC (SWI &5B)

Calculate the cyclic-redundancy check for a block of data

On entry

R0 = CRC continuation value, or zero to start
R1 = pointer to start of block
R2 = pointer to end of block
R3 = increment (in bytes)

On exit

R0 = CRC calculated
R1 - R3 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI calculates the cyclic-redundancy check value for a block of data. This is used to check for errors when, for example, a block of data is stored on a disk (although ADFS doesn't use this call) or is sent across a network and so on. If the CRC calculated when checking the block is different from the old one, then there are some errors in the data.

The block described in R1 and R2 is exclusive. That is, the calculation adds R3 to R1 each step until R1 equals R2. If they never become equal, then it will continue until crashing the machine. For example R1=100, R2=200, R3=3 will never match R1 with R2 and is not permitted.

The value of the increment in R3 is the unit that you wish to use for each step of the CRC calculation. Usually, it would be 1, 2 or 4 bytes, but any value is permitted. Note that the increment can be negative if you require it.

Related SWIs

None

Related vectors

None

IIC_Control (SWI &240)

Control IIC devices

On entry

R0 = device address (bit 0 = 0 to write, bit 0 =1 to read)
R1 = pointer to block
R2 = length of block in bytes

On exit

R0 - R2 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 allows reading and writing to IIC devices. IIC is an internal serial protocol. It is used in RISC OS machines for writing to the clock chip and IIC compatible devices on expansion cards.

The possible error is ‘No acknowledge from IIC device’ (&20300).

Related SWIs

None

Related vectors

None

*Commands

*Configure Language

Sets the configured module that will be run as an application at power on

Syntax

`*Configure Language` *`module_no`*

Parameters

`module_no`	the number of the module which will be run as an application at power on. The default is the desktop.

Use

*Configure Language sets the configured module that will be run as an application at power on. The module is specified by its module number, as returned by the *Modules command. You should use that command to check the number you're configuring, especially if you have added or removed modules. You should also be aware that module numbers may differ between versions of RISC OS.

Note that the Desktop module checks on entry if a file – such as a boot file – is being run using *Exec, and if so exits. This means your machine will fail to enter the desktop if you've configured your machine both to use the desktop as its start up language, and to run a boot file at power-on using *Exec. The cure is either to select the desktop at the end of the boot file using the *Desktop command, or to configure the machine to boot from an Obey file.

The unusual use of the word 'language' in this command's name dates from earlier Acorn operating systems, and is preserved for backwards compatibility.

Example

`*Configure Language 0` *Starts up in Command Line mode, with * prompt*

Related commands

*Configure Boot, *Desktop, *Modules, *Opt 4

Related SWIs

None

Related vectors

None

*Help

Gives brief information about each command

Syntax

`*Help [keyword]`

Parameters

keyword	the command name(s) to get help on

Use

*Help gives brief information about each command in the machine operating system, including its syntax. It also has help on some special keywords:

`*Help Commands`	lists all the available utility commands
`*Help FileCommands`	lists all the commands relating to filing systems
`*Help Modules`	lists the names of all currently loaded modules, with their version numbers and creation dates
`*Help Station`	displays the current network and station numbers of your machine
`*Help Syntax`	explains the format used for syntax messages

The usual use of *Help is to confirm that a command is appropriate for the job required, and to check on its syntax (the number, type and ordering of parameters that the command requires). When you issue the *Help command at the normal Command Line prompt, ‘paged mode’ is switched on: the computer displays a screenful of text, then waits until you press Shift before moving on.

Example

The specification of the keyword can include abbreviations to allow groups of commands to be specified. For example,

`*Help Con.`	*produces information on *Configure and *Continue*
`*Help .`	*gives help on all subjects*

Related commands

None

Related SWIs

None

Related vectors

None