About this manual

Summary of contents

This manual gives you detailed information on RISC OS 3.5 and RISC OS 3.6, so that you can write programs to run on Acorn computers that use them. It must be used in conjunction with the *RISC OS 3 Programmer's Reference Manual*, and is produced as a replacement for the earlier volume 5 in the set that described RISC OS 3.5 only. The pages are numbered '5a-*n*' rather than '5-*n*' to distinguish references to the two different versions.

This manual only tells you about the differences between RISC OS 3.1, RISC OS 3.5 and RISC OS 3.6. Many cross references are given between this volume and the earlier volumes so that you can always refer to the main topic to obtain further information.

The layout of chapters

We've laid out the information in this manual 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.

Appendix C: Errata and omissions for RISC OS 3 PRM

This appendix (on page 5a-667) contains a list of errata and omissions for the *RISC OS 3 Programmer's Reference Manual*. We suggest you add to your copy either the corrections themselves, or a reference to them.

Indexes

The separate volume of Indexes replaces that supplied with the *RISC OS 3 Programmer's Reference Manual*, and references all five volumes. It contains:

- an index of * Commands
- an index of OS_Byte calls
- an index of OS_Word calls
- a numeric index of service calls
- an alphabetic index of service 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.
- 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 3 User 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 C/C++* product, which you can use to write programs in C, C++, and ARM assembler. The product runs in a desktop environment with full supporting tools. It also provides the User Interface Toolbox, making it much easier to design and code a desktop application's user interface; for more details see *The Toolbox modules* on page 5a-657.

Technical Reference Manuals are available for all but the oldest of Acorn RISC OS computers. These describe the hardware in full, including such things as parts lists and circuit diagrams.

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 15 – The kernel

98 Introduction to RISC OS 3.5 and RISC OS 3.6

Introduction

RISC OS 3.5 is an operating system written by Acorn for its Risc PC computers that use the new ARM600 / ARM700 hardware architecture. This version was only changed where it was necessary to support the changing hardware. RISC OS 3.6 is a further development, which adds support for machines using the similar ARM7500 architecture, and integrates software that was previously separately available. We have tried to make both versions as compatible as possible with the RISC OS 3.1 operating system.

RISC OS terminology

The operating system known as RISC OS 2 in this manual consists of two variants, RISC OS 2.00 and RISC OS 2.01.

The operating system known as RISC OS 3 in this manual consists of two variants, RISC OS 3.00 and RISC OS 3.10.

The operating system known as RISC OS 3.5 in this manual is RISC OS 3.50, and is the version supplied with the first generation of Risc PC computers.

The operating system known as RISC OS 3.6 in this manual is RISC OS 3.60, and is the version supplied with the second generation of Risc PC computers, and the first generation of A7000 computers.

Hardware overview

The main electronic components of a Risc PC computer are:

- An ARM (Advanced RISC Machines) ARM610 or ARM700 processor, which provides the main processing.
- A VIDC20 (Video Controller) chip, which provides the video and sound outputs.
- An IOMD (Input Output Memory Device) which provides the interface between the ARM chip, the VIDC chip, the memory and other support chips.
  This chip replaces the IOC and MEMC chips used in earlier RISC OS computers.

The main component of the A7000 is the ARM7500 chip; this integrates all the above functionality into a single chip.

Other components

The other components are:

- ROM (Read Only Memory) chips containing the operating system.
- RAM (Random Access Memory) chips.
- VRAM (Video RAM) chips used for video display (if fitted).
- Peripheral controllers (for devices such as discs, the serial port, networks and so on).

Schematic

The diagram on page 5a-5 gives a schematic of an architecture which may be viewed as typical of the Risc PC range of computers.

ARM 610 and ARM 700

The ARM is a RISC (Reduced Instruction Set Computer) processor. The initial range of Risc PC computers can use two different versions of the ARM processor.

- The ARM610 delivers about 5 times the power of an ARM 2 (23 MIPs, or million instructions per second, compared to some 4 - 5 MIPS for the ARM2).
- The ARM700 delivers about 8 times the power of an ARM 2 (an estimated 35 MIPS). The ARM 700 also has a direct connection for a hardware floating point chip.

From the application programmer’s point of view, there is no difference between the two processors. The ARM700 supports the same instruction set as the ARM610.

It is possible that other chips in the ARM6 / ARM7 family may also be used.

The VIDC20 chip

The VIDC20 chip is an updated version of the VIDC1 and VIDC1a chip used in the previous generation of Acorn computers. The main differences are that VIDC20 provides:

- A wide range of resolution options including VGA, SuperVGA and XGA resolution.
- 1, 2, 4, 8, 16 and 32 bits per pixel.
- 8 bit DACs giving 16 million colours.

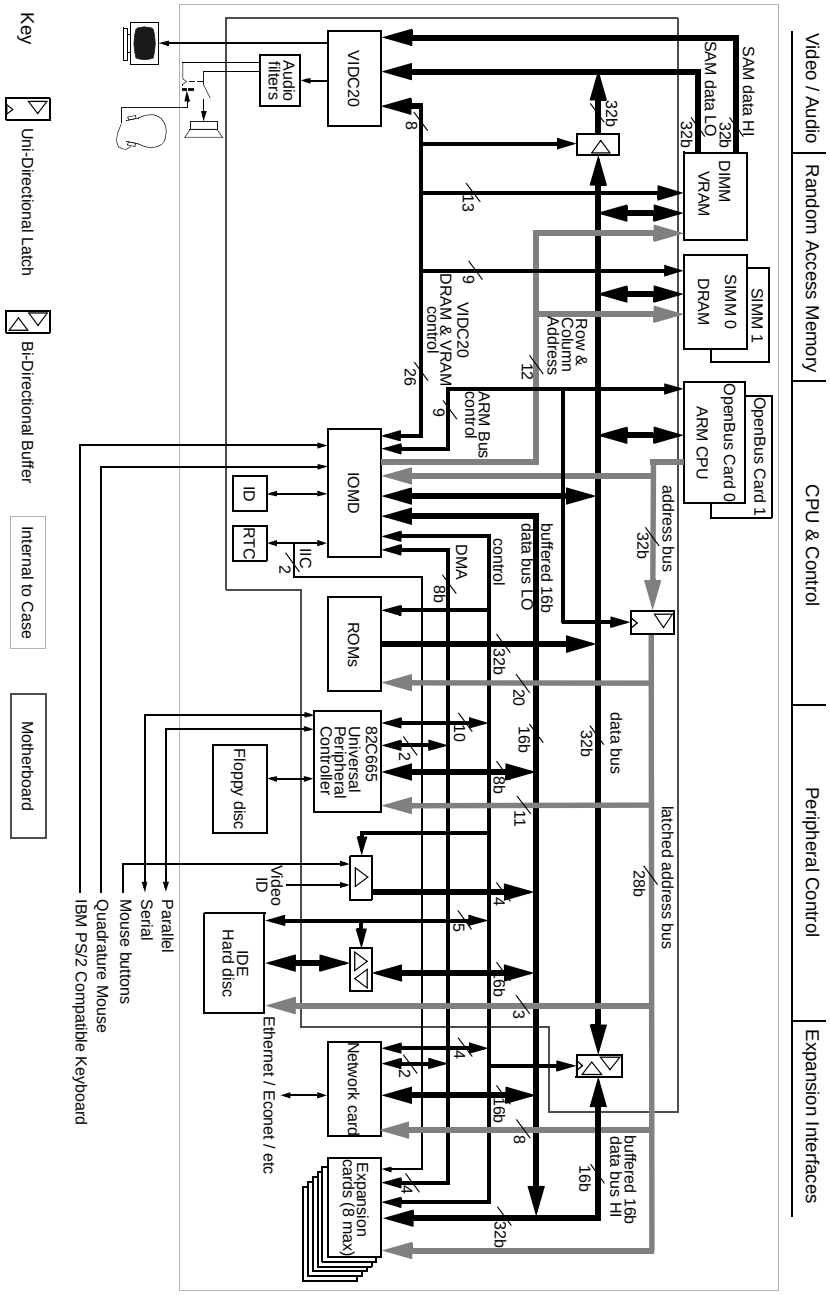


Figure 98.1 Schematic of a Risc PC computer

Video data transfer

The VIDC20 has a 64 bit data bus allowing a high data bandwidth from memory. VIDC20 takes data from the memory banks under DMA control. VIDC20 takes its data from VRAM if it is fitted, otherwise it takes data from DRAM.

Palette

The VIDC 20 contains 296 write-only registers: 256 of these are used as the 28 bit video palette entries. Each entry uses 8 bits for Red, 8 bits for Green and 8 bits for Blue with 4 bits for external data.

The video palette entries or Look up tables (LUT) allow for logical to physical translation and gamma correction. The Red, Green and Blue LUTs each drive their respective DACs. These DACs give a total of 16 million possible colours.

Pixel clock

VIDC20 can generate a display at any pixel rate up to at least 110Mhz. The clock can be selected from one of three sources, and then divided by a factor of between 1 and 8.

The VIDC20 also contains a phase comparator which – when used with an external Voltage Controlled Oscillator – forms a Phase Locked Loop. This allows a single reference clock to generate all the required frequencies for any display mode. You do not need multiple external crystals.

Sound system

The sound system is compatible with the VIDC1 sound system with an independent sound clock (24MHz). It features an 8 bit (logarithmic) system using an internal DAC. This gives eight channels each with its own stereo position.

The device can work with 1, 2, 4 or 8 stereo channels using time division multiplexing to synthesise left and right outputs. The sample rate is programmable through the Sound Frequency Register.

Cursor

VIDC20 has a hardware cursor for all its modes. The cursor is 32 pixels wide and any number of pixels high. Each pixel can be transparent, or one of three colours chosen from its own 28 bit wide palette. The cursor can be any shape or colour within these limits.

The IOMD chip

The IOMD is a specialised custom chip that takes the place of several large chips used in the old architecture.

IOMD includes some of the circuitry formerly in the IOC and MEMC chips, as well as a large amount of ‘glue’ logic.

The features of the IOMD include:

- Direct interface to ARM6xx/ARM7xx processors
- 16 bit steered bus, for on-board peripherals
- IOC functionality (ticker, interrupt manager, IIC, I/O control)
- Memory controller for DRAM and VRAM
- DMA controller for I/O, sound, cursor and video data
- PC keyboard interface
- Quadrature mouse interface.

General architecture

The IOMD is a memory, DMA and I/O controller.

It has a CPU interface for an ARM610/ARM700 type processor which can allow an additional processor to be connected. The CPU interface consists of the processor address, data and control buses.

There is a DRAM and VRAM control bus which has RAS, CAS, multiplexed address and other control lines. There are a number of DMA address generators, for sound, cursor, and general I/O DMA. There is also VRAM control logic, including logic to generate transfer cycles.

Since the whole 32 bits of the main system bus connects to IOMD, it is possible for IOMD to transfer data using DMA (Direct Memory Access) from DRAM into itself. There is a 16 bit I/O bus on IOMD, and there is byte (and half-word) steering logic to allow DMA data at arbitrary byte (or half-word) memory locations to be transferred to/from the I/O system using this bus. The 16 bit I/O bus forms the lower 16 bits of the 32 bit podule interface. IOMD controls the latches for the upper 16 bits of the extended podule bus, which allows 32 bit transfers.

IOMD contains a large subset of the functionality of IOC, including two general purpose counter/timers (timer 0 and timer 1) and the interrupt control registers. The IOC baud rate and keyboard serial rate timers are not implemented in IOMD, nor are all of the general purpose I/O lines. The allocation of interrupt lines is largely similar to previous machines.

IOMD provides a PC keyboard interface instead of the Archimedes KART interface supported by IOC. This consists of an 8 bit synchronous serial interface, with interrupt generation capability.

The chip contains a quadrature mouse interface. This consists of X and Y counters that are incremented and decremented by mouse movements. The counters wrap when they overflow or underflow, and are read regularly under interrupt. The VSync interrupt is used (although the centi-second timer could be used) as it allows updating every frame; there is no point in updating the screen more often than this. The X and Y counters are each 16 bits wide.

ARM7500

The ARM7500 is a monolithic device that integrates an ARM7 processor, a video generator similar to VIDC20, and most of the functions of IOMD. The major differences are:

- The ARM7500 provides two PS/2-style asynchronous serial keyboard ports (one for the keyboard, and one for the mouse), rather than IOMD's synchronous serial keyboard port and quadrature mouse interface of IOMD.
- The ARM7500 provides a four channel PC joystick interface, not available with IOMD.

RISC OS overview

The chapters that follow describe the changes introduced in RISC OS 3.5 and RISC OS 3.6. These changes are summarised below.

RISC OS 3.5

- Memory management has been considerably improved. Much greater amounts of physical memory are supported, and the address space is larger. Second processors can claim memory. You can now create and manipulate your own dynamic areas.
- A module has been provided to support DMA (direct memory access).
- Video and sprite capabilities have been extended to support the huge range of screen modes and colours now possible. There are new ways of selecting and specifying screen modes and monitors, and a new sprite format. Many calls have been extended to support these.
- The parallel and serial device drivers have been made considerably more fast and efficient.
- The buffer manager now allows you to insert and remove buffered data without incurring the overheads of calling SWIs

- Keyboard support has been removed from the kernel. It has been replaced by a device driver module so a standard PC keyboard can be used, greatly expanding the range of available input devices.
- The quadrature mouse driver has been removed from the kernel and placed in its own module. A serial mouse driver for a PC-type mouse is available as an alternative.
- The CMOS RAM and hard disc can now be password protected against malicious or accidental changes. The CMOS RAM can also be protected in hardware against the effects of power-on resets.
- The reset behaviour has been rationalised.
- Support is provided in ROM for AUN (*Acorn Universal Networking*).
- The appearance of the desktop has been considerably improved. It now has a 3D appearance, uses an outline font, and can tile window backgrounds with a texture.
- The Filer now allows much longer filenames, and changes column widths to accommodate them. A new icon is used to distinguish open directories. Dragged objects now appear as an icon, rather than as a dashed rectangle.
- The Wimp's error system has been extended to improve its appearance, allow more customisation, and provide more user friendly dialogues.
- DragASprite now makes dragged icons semi-transparent by default, so you can easily see what lies under them.
- A watchdog has been added so you can easily kill runaway programs that do not return control to the Wimp.
- A new Boot application has been added. Your applications can easily add and remove commands to this application, making their installation and removal much easier.
- A new ColourPicker module provides a facility for all applications to use when colours must be specified. It of course supports the full range of colours available under RISC OS 3.5.
- Expansion cards now have 32 bit wide data paths, and a directly mapped area of 16MB per card.
- A new dedicated network interface is supported.
- Screen blanking now supports monitor power saving using the new DPMS standard.

RISC OS 3.6

- Modules can use a message file when outputting text from the help and command keyword table.

- Further minor extensions have been made to the video system: in particular, support has been added for palettes in the new sprite format.
- The SpriteExtend module's SWI interface has been extended to support JPEG images, providing information on the images, and simple scaled plotting and printing.
- The new CompressJPEG module provides SWIs with which you can compress raw data into a JPEG image.
- The Draw file format has been extended so you can include JPEG images.
- FileSwitch, FileCore and the Free module have been extended to support larger capacity storage devices. Under FileCore, the recommended maximum hard disc size is 4 GB, and the maximum size of a file (and hence of an image filing system) is 2 GB.
- ADFS supports IDE discs that use logical block addressing – a method of disc addressing which is superseding the old cylinder-head-sector method.
- The 32 MB limit on the size of a DOSFS image file has been removed by using a newer type of DOS boot block. DOSFS is also less stringent in its checking of DOS formats; some discs that earlier versions of DOSFS rejected are now accepted.
- Support has been added for CDs and CD-ROMs. The CDFS filing system can access files on a CD-ROM that conforms to the widely used ISO 9660 standard. There are commands with which you can play audio CDs and read audio data directly from a CD.
- The keyboard and mouse drivers support a PS/2 keyboard and mouse, using an asynchronous serial interface such as the ARM7500 provides.
- The cut-down Internet module provided as part of RISC OS 3.5's AUN support has been replaced by the complete version.
- Acorn Access – Acorn's entry level product for AUN networking – is now a part of RISC OS. It provides peer to peer networking using TCP/IP protocols, allowing sharing of resources such as discs and printers.
- The new DragAnObject module provides SWI calls similar to those provided by the DragASprite module, save that you can use them to make the pointer drag any object that you can render.
- The new DrawFile module renders Draw files either to the screen, or to a printer driver during printing. This makes it easy for you to support imported Draw files in your applications.
- The range of Boot applications has been extended, mainly to support network booting.
- Further changes have been made to printing, largely to support JPEGs.

- The SoundDMA module has been extended to support 16 bit sound, as well as the 8 bit μ-law sound used by all earlier versions of RISC OS.
- The Joystick module has been extended both to support PC-style analogue joysticks, and to provide calls used with analogue input devices on older Acorn machines.
- The Toolbox modules from Acorn C/C++ have been added to RISC OS. Toolbox applications therefore don't need to load the modules into RAM, hence decreasing their memory usage.

99 ARM hardware

Introduction and Overview

The ARM architecture changed significantly with the introduction of the ARM6 series. The section below describes the **differences** in behaviour of more recent ARM processors, used with RISC OS 3.5 and later. For details of earlier ARM processors, see the chapter entitled *ARM Hardware* on page 1-9.

32 bit architecture

New features in ARM6

The most notable change made in the ARM6 series was to extend the address bus and program counter to a full 32 bits. As a result:

- The PSR had to be separated from the PC into its own register, the CPSR (*Current Program Status Register*).
- The PSR can no longer be saved with the PC when changing processor modes; instead, each privileged mode now has an extra register – the SPSR (*Saved Program Status Register*) – to hold the previous mode's PSR.
- Instructions have been added to use these new status registers.

A further change was the addition of extra privileged processor modes, allowed by the PSR now having a full 32 bits to use. These modes are used to handle Undefined instruction and Abort exceptions. Consequently:

- Undefined instructions, aborts, and supervisor code no longer have to share the same mode. This has removed restrictions on Supervisor mode programs which existed on earlier ARMs.

Processor configuration

The availability of these features in the ARM6 series (and other later compatible chips) is set by one of several on-chip control registers. One of three *processor configurations* can be selected:

- **26 bit program and data space**. This configuration forces ARM to operate with a 26 bit address space. In this configuration only the four 26 bit modes are available (see *Processor modes* below); it is impossible to select a 32 bit mode.

  This configuration is set at reset on all current ARM6 and 7 series processors.
- **26 bit program space and 32 bit data space.** This is the same as the 26 bit program and data space configuration, except that address exceptions are disabled to allow data transfer operations to access the full 32 bit address space.
- **32 bit program and data space**. This configuration extends the address space to 32 bits, and introduces major changes to the programmer's model. In this configuration you can select any of the 26 bit and the 32 bit processor modes (see *Processor modes* below).

Processor modes

When configured for a 32 bit program and data space, the ARM6 and ARM7 series support ten overlapping *processor modes* of operation:

- User mode: the normal program execution state – or

  User26 mode: a 26 bit version of the above
- FIQ mode: designed to support a data transfer or channel process – or

  FIQ26 mode: a 26 bit version of the above
- IRQ mode: used for general purpose interrupt handling – or

  IRQ26 mode: a 26 bit version of the above
- SVC mode: a protected mode for the operating system – or

  SVC26 mode: a 26 bit version of the above
- Abort mode (abbreviated to ABT mode): entered after a data or instruction prefetch abort
- Undefined mode (abbreviated to UND mode): entered when an undefined instruction is executed.

The distinction between processor **modes** and **configurations** is important, and will be rigidly adhered to in the rest of this manual.

The 26 bit processor modes

When in a 26 bit processor mode, the programmer's model reverts to that of earlier 26 bit ARM processors. The behaviour is the same as that of the ARM2aS macrocell with the following alterations:

- Address exceptions are only generated by ARM when it is configured for 26 bit program and data space.

  In other configurations the OS may still simulate the behaviour of address exception, using external logic such as a memory management unit to generate an abort if the 64Mbyte range is exceeded, and converting that abort into an 'address exception trap' for the application.
- The new instructions to transfer data between general registers and the program status registers remain operative. The new instructions can be used by the operating system to return to a 32 bit mode after calling a binary containing code written for a 26 bit ARM.
- When in a 32 bit program and data space configuration, all exceptions (including Undefined Instruction and Software Interrupt) return the processor to a 32 bit mode, so the operating system must be modified to handle them.
- If the processor attempts to write to a location between &0 and &1F inclusive (i.e. the exception vectors), hardware prevents the write operation and generates a data abort. This allows the operating system to intercept all changes to the exception vectors and redirect the vector to some veneer code. The veneer code should place the processor in a 26 bit mode before calling the 26 bit exception handler.

In all other respects, when operating in a 26 bit mode the ARM behaves as like a 26 bit ARM. (See the chapter entitled *ARM Hardware* on page 1-9.) The relevant bits of the CPSR appear to be incorporated back into R15 to form the PC/PSR with the I and F bits in bits 27 and 26. The instruction set behaves like that of the ARM2aS macrocell, with the addition of the MRS and MSR instructions.

RISC OS processor configuration and modes

Early in its startup code, RISC OS writes to the ARM's control register to change it into the 32 bit program and data space configuration, where it remains. You must not alter the processor's configuration yourself when writing code for RISC OS.

Although RISC OS runs under a 32 bit configuration, it remains in 26 bit modes for normal operation, providing a high degree of backward compatibility with code written to run on earlier 26 bit processors.

However, because the processor is in a 32 bit configuration, all exceptions (including Undefined Instruction and Software Interrupt) force the processor to a privileged 32 bit mode appropriate to the exception. There are therefore some differences in exception handling between 26 and 32 bit architecture ARM chips, although RISC OS provides a

considerable degree of backward compatibility by faking 26 bit behaviour on 32 bit architecture chips in most circumstances. For full details, see the chapter entitled *Hardware vectors* on page 5a-21.

Registers

The registers available in the ARM6 and ARM7 series are:

User and User26 mode	SVC and SVC26 mode	IRQ and IRQ26 mode	ABT mode	UND mode	FIQ and FIQ26 mode
R0					
R1					
R2					
R3					
R4					
R5					
R6					
R7					
R8					R8_fiq
R9					R9_fiq
R10					R10_fiq
R11					R11_fiq
R12					R12_fiq
R13	R13_svc	R13_irq	R13_abt	R13_und	R13_fiq
R14	R14_svc	R14_irq	R14_abt	R14_und	R14_fiq
R15 (PC)					

CPSR					
	SPSR_svc	SPSR_irq	SPSR_abt	SPSR_und	SPSR_fiq

Figure 99.1 32 bit register organisation

These are similar to those available in the ARM2 and ARM3 series registers. The key differences are:

- the PC is a full 32 bits wide
- the PSR is held in its own register, the CPSR (see the section entitled *The CPSR and SPSR registers* below)
- each privileged mode has a private SPSR register in which to save the CPSR
- there are two new privileged modes, each of which has private copies of R13 and R14.

The CPSR and SPSR registers

The allocation of the bits within the CPSR (and the SPSR registers to which it is saved) is shown in the figure *The Current Process Status Register (CPSR)* below.

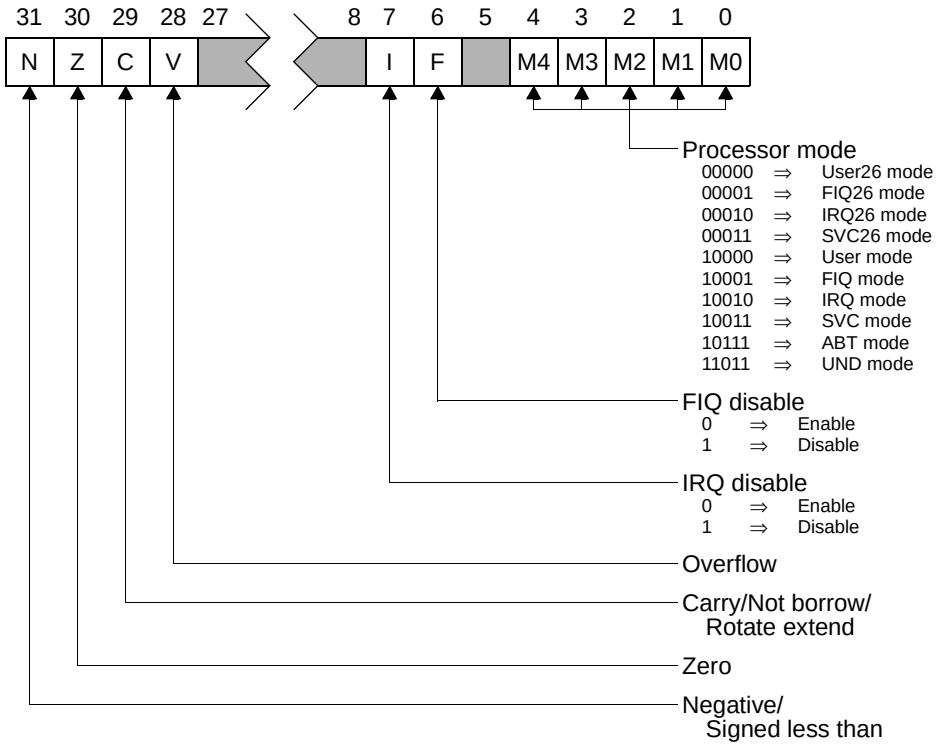


Figure 99.2 The Current Process Status Register (CPSR)

Block diagram of core

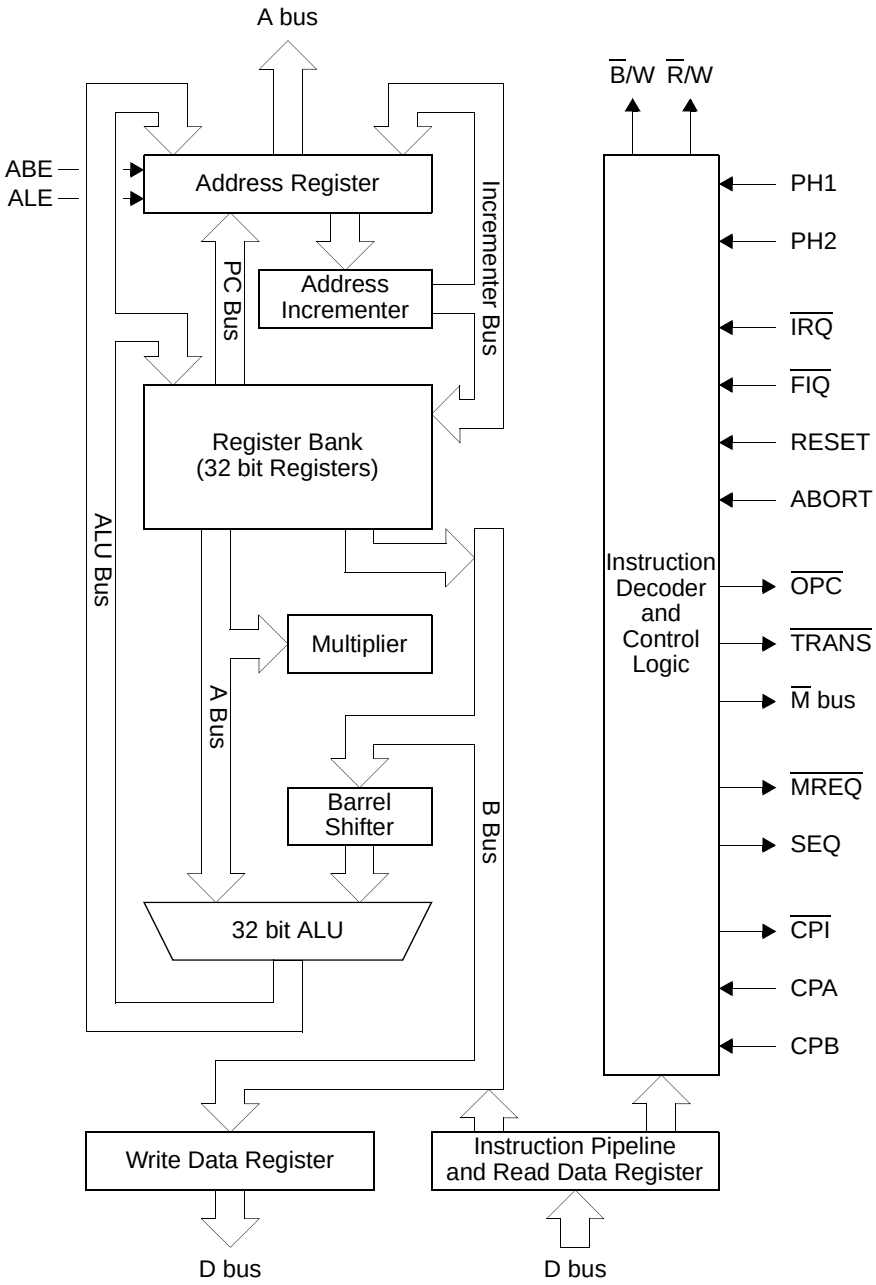


Figure 99.3 ARM Core block diagram

Block diagram of core

100 Hardware vectors

Introduction and Overview

This chapter describes the ways in which the 32 bit processor configuration used by RISC OS 3.5 and later affects exception handling. If you are writing any exception handler, you must read both this chapter and the chapter entitled *Hardware vectors* on page 1-113.

Exceptions

Introduction

Exceptions arise whenever there is a need for the normal flow of program execution to be broken, so that (for instance) the processor can be diverted to handle an interrupt from a peripheral. The processor state just prior to handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. Many exceptions may arise at the same time.

ARM handles exceptions by making use of the banked registers to save state. The old PC and PSR are copied, under RISC OS 3.1 or earlier (ie on a 26 bit addressing ARM) to the appropriate R14, or under RISC OS 3.5 or later (ie on a 32 bit configured ARM) to the appropriate R14 and SPSR. The PC and processor mode bits are forced to a value which depends on the exception. Interrupt disable flags are set where required to prevent otherwise unmanageable nestings of exceptions. In the case of a re-entrant interrupt handler, R14 should be saved onto a stack in main memory before re-enabling the interrupt. When multiple exceptions arise simultaneously a fixed priority determines the order in which they are handled.

FIQ (Fast interrupt request)

The FIQ (Fast Interrupt reQuest) exception is externally generated by taking the $\overline{\text{FIQ}}$ pin LOW. This input can accept asynchronous transitions, and is delayed by one clock cycle for synchronisation before it can affect the processor execution flow. It is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, so that the overhead of context switching is minimised.

The FIQ exception may be disabled by setting the F flag in the PSR (but note that this is not possible from User mode). If the F flag is clear ARM checks for a LOW level on the output of the FIQ synchroniser at the end of each instruction.

When ARM is successfully FIQed it will:

1 Save R15 in R14_fiq, and (for RISC OS 3.5 or later) save the CPSR in SPSR_fiq.

2 Force the mode bits to FIQ mode and set the F and I bits in the PSR.

3 Force the PC to fetch the next instruction from address &1C.

To return normally from FIQ use:

```
SUBS PC,R14_fiq,#4
```

This will resume execution of the interrupted code sequence, and restore the original mode and interrupt enable state.

IRQ (Interrupt request)

The IRQ (Interrupt ReQuest) exception is a normal interrupt caused by a LOW level on the $\overline{\text{IRQ}}$ pin. This input can accept asynchronous transitions, and is delayed by one clock cycle for synchronisation before it can affect processor execution. It has a lower priority than FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the PC (but note that this is not possible from user mode). If the I flag is clear ARM checks for a LOW level on the output of the IRQ synchroniser at the end of each instruction.

When ARM is successfully IRQed it will:

1 Save R15 in R14_irq, and (for RISC OS 3.5 or later) save the CPSR in SPSR_irq.

2 Force the mode bits to IRQ mode and set the I bit in the PSR.

3 Force the PC to fetch the next instruction from address &18.

To return normally from IRQ use:

```
SUBS PC,R14_irq,#4
```

This will restore the original processor state and thereby re-enable IRQ.

Address exception trap

Under RISC OS 3.5 or later, address exceptions are **never** generated, and you may therefore ignore this section.

Under RISC OS 3.1 or earlier, an address exception arises whenever a data transfer is attempted with a calculated address above &3FFFFFF. The ARM address bus is 26 bits wide, but an address calculation has a 32 bit result. If this result has a logic '1' in any of the top 6 bits it is assumed that the address overflow is an error, and the address exception trap is taken.

Note that a branch cannot cause an address exception, and a block data transfer instruction which starts in the legal area but increments into the illegal area will not trap (it wraps round to address 0 instead). The check is performed only on the address of the first word to be transferred.

When an address exception is seen ARM will:

1. If the data transfer was a store, force it to load. (This protects the memory from spurious writing.)
2. Complete the instruction, but prevent internal state changes where possible. The state changes are the same as if the instruction had aborted on the data transfer.
3. Save R15 in R14_svc.
4. Force the mode bits to SVC mode and set the I bit in the PSR.
5. Force the PC to fetch the next instruction from address &14.

Normally an address exception is caused by erroneous code, and it is inappropriate to resume execution. If a return is required from this trap, use `SUBS PC,R14_svc,#4`. This will return to the instruction after the one causing the trap.

Abort

The Abort signal comes from an external Memory Management system, and indicates that the current memory access cannot be completed. ARM checks for an Abort at the end of the first phase of each bus cycle. When successfully Aborted ARM will respond in one of three ways.

Abort during instruction prefetch

If abort is signalled during an instruction prefetch (a *Prefetch abort*), the prefetched instruction is marked as invalid; when it comes to execution, it is reinterpreted as below. (If the instruction is not executed, for example as a result of a branch being taken while it is in the pipeline, the abort will have no effect.)

Then ARM will:

1. Save R15 in R14_svc, or (for RISC OS 3.5 or later) save R15 in R14_abt and save the CPSR in SPSR_abt.
2. Force the mode bits to SVC mode or (for RISC OS 3.5 or later) ABT mode and set the I bit in the PSR.

3 Force the PC to fetch the next instruction from address &0C.

To continue after a Prefetch abort use `SUBS PC,R14,#4` (where `R14` is `R14_svc` or `R14_abt` depending on the version of RISC OS). The ARM will then re-execute the aborting instruction, so you should ensure that you have removed the cause of the original abort.

Abort during data access

If the abort command occurs during a data access (a *Data Abort*), the action depends on the instruction type.

- Single data transfer instructions (LDR and STR) are aborted as though the instruction had not executed.
- Block data transfer instructions (LDM and STM) complete, and if writeback is set, the base is updated. If the instruction would normally have overwritten the base with data (ie LDM with the base in the transfer list), this overwriting is prevented. All register overwriting is prevented after the Abort is indicated, which means in particular that R15 (which is always last to be transferred) is preserved in an aborted LDM instruction.

Then ARM will:

1 Save R15 in R14_svc, or (for RISC OS 3.5 or later) save R15 in R14_abt and save the CPSR in SPSR_abt.

2 Force the mode bits to SVC mode or (for RISC OS 3.5 or later) ABT mode and set the I bit in the PSR.

3 Force the PC to fetch the next instruction from address &10.

To continue after a data abort, remove the cause of the abort, then reverse any auto-indexing that the original instruction may have done, then return to the original instruction with `SUBS PC,R14,#8` (where `R14` is `R14_svc` or `R14_abt` depending on the processor configuration).

Abort during an internal cycle

The ARM ignores aborts signalled during internal cycles.

Using aborts to implement virtual memory systems

The abort mechanism allows a ‘demand paged virtual memory system’ to be implemented when a suitable memory management unit (such as MEMC) is available. The processor is allowed to generate arbitrary addresses, and when the data at an address is unavailable the memory manager signals an abort. The processor traps into system software which must work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

Software interrupt

The software interrupt instruction is used for getting into supervisor mode, usually to request a particular supervisor function. ARM will:

1. Save R15 in R14_svc, and (for RISC OS 3.5 or later) save the CPSR in SPSR_svc.
2. Force the mode bits to SVC mode and set the I bit in the PSR.
3. Force the PC to fetch the next instruction from address &8.

To return from a SWI, use `MOVS PC,R14_svc`. This returns to the instruction following the SWI.

Undefined instruction trap

When ARM executes a coprocessor instruction or an undefined instruction, it offers it to any coprocessors which may be present. If a coprocessor can perform this instruction but is busy at that moment, ARM will wait until the coprocessor is ready. If no coprocessor can handle the instruction ARM will take the undefined instruction trap.

When the undefined instruction trap is taken ARM will:

1. Save R15 in R14_svc, or (for RISC OS 3.5 or later) save R15 in R14_und and save the CPSR in SPSR_und.
2. Force the mode bits to SVC mode or (for RISC OS 3.5 or later) UND mode and set the I bit in the PSR.
3. Force the PC to fetch the next instruction from address &4.

The undefined instruction trap may be used for software emulation of a coprocessor in a system which does not have the coprocessor hardware; or for general purpose instruction set extension by software emulation (the floating point instruction set is implemented in software this way).

To return from this trap (after performing a suitable emulation of the required function), use `MOVS PC,R14` (where `R14` is `R14_svc` or `R14_und` depending on the processor configuration). This will return to the instruction following the undefined instruction.

Reset

ARM can be reset by pulling its RESET pin HIGH. If this happens, ARM will stop the currently executing instruction and start executing no-ops. When RESET goes LOW again, it will:

1. Save R15 in R14_svc, and (for RISC OS 3.5 or later) save the CPSR in SPSR_svc.
2. Force the mode bits to SVC mode and set the F and I bits in the PSR.
3. Force the PC to fetch the next instruction from address &0.

Vector summary

The first eight words of store normally contain branch instructions pointing to the relevant routines. The FIQ routine may reside at &000001C onwards, and thereby avoid the need for (and execution time of) a branch instruction.

Address	Definition
&0000000	Reset
&0000004	Undefined instruction
&0000008	Software interrupt
&000000C	Abort (prefetch)
&0000010	Abort (data)
&0000014	Address exception
&0000018	IRQ
&000001C	FIQ

Exception Priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they will be handled:

1 Reset (highest priority)

2 Address exception, Data abort

3 FIQ

4 IRQ

5 Prefetch abort

6 Undefined Instruction, Software interrupt (lowest priority)

Note that not all exceptions can occur at once. Address exception and data abort are mutually exclusive, since if an address is illegal the ARM will ignore the ABORT input. Undefined instruction and software interrupt are also mutually exclusive since they each correspond to particular (non-overlapping) decodings of the current instruction.

If an address exception or data abort occurs at the same time as a FIQ, and FIQs are enabled (ie the F flag in the PSR is clear), ARM will enter the address exception or data abort handler and then immediately proceed to the FIQ vector. A normal return from FIQ will cause the address exception or data abort handler to resume execution. Placing address exception and data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection, but the time for this exception entry should be added to worst case FIQ latency calculations.

The pre-veneers

To ensure easy backward compatibility, versions of RISC OS from 3.5 onwards install a *pre-veneer* on all hardware vectors apart from FIQ (see the section entitled *Writing to the FIQ vector* on page 5a-28) and address exception (which is never generated by a 32 bit configured ARM). Each pre-veneer first sets up R14 to contain a combined PC and PSR that will return the processor to the 26 bit mode it was in when the exception arose. It then places the processor in the privileged 26 bit mode used by the earlier 26 bit chips for that exception. It thus effectively fakes the behaviour of earlier versions of RISC OS that run on those chips.

The pre-veneer is called before any exception handlers that are installed with software interfaces such as OS_ChangeEnvironment, so you can usually use such exception handlers unchanged on all versions of RISC OS (hardware dependencies excepted).

Entering 32 bit modes

One consequence of this is that **you may not enter a 32 bit mode with IRQs enabled**. Were you to do so, and an IRQ were to occur, the IRQ pre-veneer would be entered; then the IRQ handler would return you to a 26 bit mode, rather than the 32 bit mode you were in at the time of the IRQ.

Note that you shouldn't use 32 bit modes except for writing exception handlers; see the section entitled *Running 32 bit code* on page 5a-37.

Claiming the hardware vectors

Under earlier versions of RISC OS, you could also claim the hardware vectors directly, by overwriting the existing instruction on the vector, and replacing it afterwards. It was your responsibility to do any housekeeping, in particular checking for subsequent claimants before restoring the original instruction.

Under 32 bit aware versions of RISC OS, if you attempt to write to any hardware vector other than FIQ a data abort is generated. You must instead call the new SWI OS_ClaimProcessorVector (page 5a-30), passing it the address of your exception handler. The handler is installed on the vector, and is called directly, before the pre-veneers. Such handlers are therefore entered in a 32 bit mode.

For handlers installed directly on the vector to work across all versions of RISC OS, you must therefore change the method of claiming and releasing the vector depending on the version of RISC OS:

- On versions up to RISC OS 3.1, you must write directly to the vector, doing any appropriate housekeeping yourself
- On later versions you must call OS_ClaimProcessorVector.

Your handler must also cope with running in both 26 bit and 32 bit modes.

Writing to the FIQ vector

On a 32 bit architecture ARM, the FIQ vector is entered in FIQ mode (i.e. the 32 bit form of the mode). There are no pre-veneers to simulate 26 bit behaviour. To install a FIQ handler, you must write directly to the FIQ vector, just as always.

The sample code below is the recommended way to write to the FIQ vector on both 26 and 32 bit configured processors – you can use the same code on all versions of RISC OS. Obviously the handler you install must cope with running in both 26 bit and 32 bit FIQ modes. In practice this is unlikely to be a problem, and most existing handlers will run unchanged once installed.

In the code, comments that are prefixed by '`32:`' apply to a 32 bit configured processor, and comments that are prefixed by '`26:`' apply to a 26 bit configured processor.

```
; We assume that at this point, you are already in a privileged 26 bit mode.

        ; 26: Does not alter processor mode. Reads as follows:
        ; NOP                           ; 26: Encodes a NOP (TST Ra,R0)
        ; Push  Ra                      ; 26: Pushes entry Ra onto stack
        ; ORR   Ra, Ra, #2_11000000     ; 26: Corrupts Ra
        ; NOP                           ; 26: Encodes a NOP (TEQ R9,Ra)
        ; ORR   Ra, Ra, #2_10000        ; 26: Corrupts Ra
        ; NOP                           ; 26: Encodes a NOP (TEQ R9,Rb)

        ; 32: Switch to _32 mode with IRQs and FIQs off.
        ; 32: Must switch interrupts off before switching mode as there can be
        ; 32: an interrupt after the MSR instruction but before the next one.
        MRS     Ra, CPSR_all            ; 32: Read privileged 26 bit mode,
        Push    Ra                      ; 32: and push it onto the stack
        ORR     Ra, Ra, #2_11000000     ; 32: Set IRQ and FIQ disable bits
        MSR     CPSR_all, Ra            ; 32: Disable IRQs and FIQs
        ORR     Ra, Ra, #2_10000        ; 32: Set M4 bit (for 32 bit mode)
        MSR     CPSR_all, Ra            ; 32: Change to 32 bit mode

        ; Now do a NOP, to let things settle down:
        NOP                             ; e.g. MOV R0,R0

; Now in a suitable mode to write FIQ handler code to FIQ vector
; (&1C-&FC incl.), whatever the processor configuration.
; Code written should be able to run in both fiq_32 and fiq_26 modes,
; and should end with a SUBS PC,R14,#4 to return normally.
; For example we might write the handler code like this:

        ; Assume Rb already points to location from which to copy the handler.

        MOV     LR, #FIQVector          ; Get address of FIQ vector

40      LDR     Ra, [Rb], #4            ; Get opcode.
        TEQS    Ra, #0                  ; All done?
        STRNE   Ra, [LR], #4            ; No, so copy to FIQ area...
        BNE     %BT40                   ; ...and repeat for next opcode.

        ; The above may not be optimal, and is for illustration only.

; Having written FIQ vector, now need to restore the original
; privileged 26 bit mode.

        ; 26: Does not alter processor mode. Reads as follows:
        ; PULL  Ra                      ; 26: Pull entry Ra from stack
        ; NOP                           ; 26: Encodes a NOP (TST Ra,R0)

        PULL    Ra                      ; 32: Pull saved CPSR, and
        MSR     CPSR_all, Ra            ; 32: Restore privileged 26 bit mode

; Now back where we started, except Ra and Rb should be treated as corrupted.
; (We must assume neither is preserved, because we don't know the processor
; configuration.)
```

SWI Calls

OS_ClaimProcessorVector (SWI &69)

Provides a means for a module to attach itself to one of the processor's vectors

On entry

R0 = vector and flags:

Bit	Meaning
0 - 7	vector number:
	0 'Branch through 0' vector
	1 Undefined instruction
	2 SWI
	3 Prefetch abort
	4 Data abort
	5 Address exception (only on ARM 2 and 3)
	6 IRQ
	(all other values reserved for future use)
8	0 ⇒ release, 1 ⇒ claim
9 - 31	reserved, must be 0

R1 = replacement address of exception handler
R2 = address which should currently be on vector (only needed for release)

On exit

R0 preserved
R1 = address which has been replaced (only returned on claim)
R2 preserved

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 means for a module to attach itself to one of the processor's vectors. This is a direct attachment; you get no environment except what the processor provides.

As such, claiming and releasing the vectors is somewhat primitive – the claims and releases must occur in the right order (the release order being the reverse of claim order). On release if the value in R2 doesn't match what is currently on the vector then an error will be returned. This ensures correct chaining of claims and releases.

Related SWIs

None

Related vectors

None

101 Interrupts

Introduction and Overview

The new IOMD or ARM 7500 chips used under RISC OS 3.5 and 3.6 use new device numbers (see *Device numbers* on page 1-120). These are:

0	Printer interrupt from controller
1	Unused
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	IDE hard disc interrupt
10	Serial port interrupt from controller – also mapped to FIQ device 4
11	Network card interrupt
12	Floppy disc interrupt from controller
13	Expansion card interrupt
14	Keyboard serial transmit register empty
15	Keyboard serial receive register full
16	DMA channel 0
17	DMA channel 1
18	DMA channel 2
19	DMA channel 3
20	DMA sound channel 0
21	DMA sound channel 1
22	ARM 7500 mouse port receive register full
23	ARM 7500 mouse port transmit register empty
24	ARM 7500 joystick A-to-D completion
25	ARM 7500 event 1
26	ARM 7500 event 2

Note that device numbers 22 - 26 are available with an ARM 7500, but not with an IOMD.

102 Modules

Introduction and Overview

Message files for help and command keyword table

RISC OS 3.6 introduces facilities for using a message file when outputting text from the help and command keyword table. This is done using a new field in the module header to specify the pathname of the message file, and a new flag in the Help and command keyword table's information word, to indicate the use of the message file.

This makes it easier to internationalise modules.

Changes to existing SWIs

OS_ServiceCall (page 1-254)

The list of service calls available in the SWI description obviously refers to RISC OS 3.1, and so is now out of date. For a complete current list, refer to the *Numeric index of Service Calls* on page Index-19 or the *Alphabetic index of Service Calls* on page Index-25.

Technical details

This section details changes introduced in RISC OS 3.6.

Module header format

The module header (see *Module header format* on page 1-208) has a new field added:

Offset	Type	Contains
&2C	offset to string	message file pathname (optional)

The string must be word-aligned and zero-terminated. It gives the pathname of a message file used when outputting text from the help and command keyword table.

Help and command keyword table

The invalid syntax and help fields of the Help and command keyword table (see *Help and command keyword table* on page 1-216) still give an offset to a string; this string is either the actual message to output, or a token to be looked up in the message file

Information word

The meaning of the invalid syntax and help strings is set by a new flag in the information word.

Bit 28 = 1

The invalid syntax and help strings are tokens, to be looked up in the message file given in the module header.

The message file

The format of the message file is as follows:

<token>:*<text><null byte><linefeed>*

The *<text>* is output by OS_PrettyPrint (page 1-536)– but in this case, rather than using the kernel dictionary, it uses the dictionary in:

`Resources:Resources.Kernel.Dictionary`

103 Memory management

Introduction

This chapter describes changes in memory management made in RISC OS 3.5. These changes have been caused by the changes in the underlying hardware used in the new architecture.

Memory management now incorporates the following:

- Up to 256MB DRAM and 2MB VRAM of memory is allowed.
- Direct memory access (DMA) control is improved.
- Any second processor card can claim a chunk of memory.
- Physical RAM allocation does not have to be contiguous.
- Page table allocation is added to support the memory management unit (MMU) in newer ARM processors.
- The logical memory map is expanded due to the 32 bit address space.

Running 32 bit code

The new generation of ARM chips used provide 26 bit processor modes (which are backwards compatible with the ARM2 and ARM3), and 32 bit processor modes. With one exception, RISC OS 3.5 only supports 26 bit processor modes. **You must not execute code in 32 bit processor modes**. If you try to do so, you may get unpredictable crashes, especially if you try to run the code in address space over 64M.

The exception mentioned above is if you are writing handlers that claim a hardware vector. For details, see the chapter entitled *Hardware vectors* on page 5a-21.

Overview

Free memory pool

In RISC OS 2 and 3 memory management was divided between the kernel and the Wimp.

- The kernel ordered the memory into dynamic areas and an application space.
- The Wimp managed a free pool and multiple applications mapped in turn into the same application space; it was responsible for constructing and managing tasks in the desktop. It grew or shrank tasks by mapping a free pool into application space above the task, and then moving the boundary between the two.

RISC OS 3.5 supports amounts of memory so large that the free pool may now be too large to map into application space.

- The kernel is therefore now responsible for managing the free pool memory, which it keeps in a new dynamic area, known as the *free pool* (area number 6).
- The Wimp's operation is simplified, as it no longer needs to maintain its own free pool.

How the free pool operates

When you grow or shrink dynamic areas other than the free pool, the free pool is used as follows:

- If an area other than the free pool is grown, memory is taken from the free pool, if any exists. The current application is not notified of this.

  If having shrunk the free pool to zero size, there is still not enough memory for the growth, the kernel attempts to remove pages from the application space as it does under existing versions of RISC OS.
- If an area other than the free pool is shrunk, the pages recovered are added to the free pool. The current application is not consulted.

The Wimp grows or shrinks tasks by shrinking or growing (respectively) the free pool itself:

- If the free pool is grown, pages are taken from application space to put in it. The current application is consulted beforehand.
- If the free pool is shrunk, the recovered pages are added to application space. The current application is consulted beforehand.

The tasks themselves must still change their memory allocation using current RISC OS interfaces (as before), rather than changing the size of the free pool.

Dynamic areas

In RISC OS 2 and 3 the main kernel interface for memory management was OS_ChangeDynamicArea (page 1-384), with which you could resize the predefined dynamic areas. This SWI then called other modules, depending on which dynamic area was being resized. This left no flexibility, and in particular, there were no facilities for creating other dynamic areas. This meant the existing areas were often used illicitly by applications which – once they quit – would leave the area badly fragmented.

Other memory related services were not available. For example it was not possible to find out what memory was available on the system without knowing a great deal about the platform.

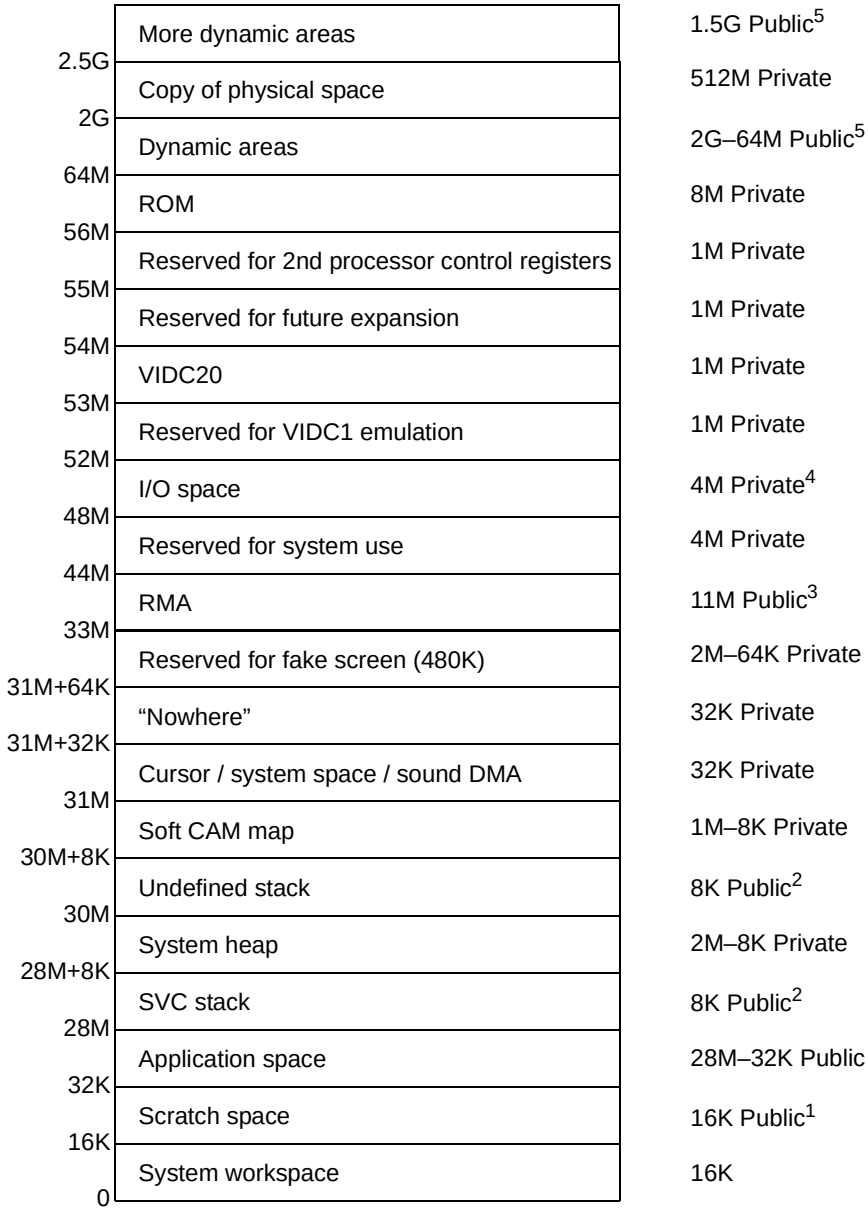
From RISC OS 3.5 onwards the new SWI OS_DynamicArea (see page 5a-53) is provided for you to create dynamic areas, get information on them, and delete them. This allows you to claim and release your own area of memory that is managed by hardware (and so does not suffer from garbage), and is persistent. This is far preferable to illicitly using (say) a part of the RMA or sprite area, as has been common practice.

You should still use OS_ChangeDynamicArea just as before to alter the size of dynamic areas.

As all operations on dynamic areas work in physical page numbers you cannot map anything other than RAM pages (DRAM and VRAM) into a dynamic area. In particular you cannot map in the extension to the existing expansion card bus space, known as the *EASI space*.

Technical Details

Logical memory map



Notes about the logical memory map:

1 The public[1] area may be used by any module that is not

- used in an IRQ routine
- used if you call something else that might also use it.

An example client would be FileCore using the scratch space to hold structures while working out how to allocate some free space. Another example would be the Filer using the scratch space to hold structures for OS_HeapSort.

2 The public[2] areas can be assumed to have their lowest address on a 1MB boundary (being descending stacks). An exception will occur if they are accessed beyond this point. The exact location of these stacks should not be assumed.

3 The public[3] area should not assume the location of the RMA or its maximum size. However it will be in the lower 64MB (ie it can execute 26 bit code).

4 The private[4] area is private, and used for I/O except where device drivers export hardware addresses.

5 The public[5] areas can be used by a client to make its own dynamic area.

Memory terminology

There are three ways of referring to memory:

Physical address

This refers to the address of the memory in the physical address space, as presented by the ARM chip to IOMD.

Logical address

This refers to the logical address space that the ARM processor core presents to the ARM chip memory management unit. This is controlled by the operating system.

Physical page number

This is an arbitrary number assigned to each page of RAM physically present in the computer.

Page blocks

Several interfaces use page blocks to pass round lists of addresses and/or pages. These are tables of 12-byte records (so a page block is 12*n* bytes long, where *n* is the number of records). Each record has the following format:

Offset	Meaning
0	Physical page number
4	Logical address
8	Physical address

New SWIs

The following new SWIs have been created. They are defined in full at the end of this chapter.

- OS_DynamicArea (page 5a-53) is used for the control, creation and deletion of dynamic areas. R0 provides a reason code which determines which operation is performed.
- OS_Memory (page 5a-62) performs miscellaneous operations for memory management. Again, R0 provides a reason code which determines which operation is performed.

Changes to existing SWIs

OS_ChangeDynamicArea (page 1-384)

You can now alter the space allocation of the *free pool* (see page 5a-38) by setting R0 to 6 on entry.

OS_SetMemMapEntries (page 1-394)

With the new architecture you must use –1 to indicate that a page should become inaccessible.

OS_ReadDynamicArea (page 1-396)

In RISC OS 3, if bit 7 of the dynamic area number is set then R2 will be returned with the maximum area size.

This has changed slightly from RISC OS 3.5 onwards.

If the dynamic area number passed in is greater than or equal to 128 then R2 is returned as the maximum size of the dynamic area. Also, if the dynamic area number passed in is between 128 and 255 inclusive then the information is returned for the area whose number is 128 less than the passed-in value.

The net result is that for old dynamic area numbers (0 - 5) the functionality is unchanged, but the number-space impact of the interface is minimised.

Also, if R0 is –1 on entry, it returns the following information on application space:

R0 = base address (&8000)
R1 = current size (ie for current task)
R2 = maximum size (≤ 28MB–32kB in current implementation)

OS_Heap 0 (page 1-377)

RISC OS 2 and 3 place strong restrictions on the heap: the base of the heap as specified in R1 must be word-aligned and less than 32Mbytes, and the size of the heap must be a multiple of 4 and less than 16Mbytes.

From RISC OS 3.5 onwards the only restrictions are that the base of the heap must be word-aligned, and the size must be a multiple of 4 bytes.

Wimp_TransferBlock (page 3-214)

In earlier operating systems Wimp_TransferBlock put all used task memory into the application space, and then copied the relevant parts over. It cannot do this any more, as the total used task memory may not fit into application space.

The algorithm used by this call has accordingly been changed, and the opportunity taken to improve its performance. The call still has the same entry and exit conditions.

Wimp_ClaimFreeMemory (page 3-208)

Because the Wimp no longer maintains control of the free pool, the call Wimp_ClaimFreeMemory has had to be modified; it simulates its previous behaviour as well as possible. In general, applications written for older versions of RISC OS will work unmodified; but you should be aware that the call may now return addresses that use more than 26 bits. This will be a problem if your old applications use any of the top 6 bits for their own purposes.

Using this call in new applications is deprecated. You should instead use OS_DynamicArea (page 5a-53) to create your own dynamic area.

Cache_... SWIs (page 4-192 onwards)

These ARM3-specific SWIs are not implemented from RISC OS 3.5 onwards.

Changes to existing * Commands

*Cache (page 4-201)

*Cache now switches both cacheing and write buffering on and off.

Dynamic area handler routines

When you create a dynamic area with the new SWI OS_DynamicArea 0 (see page 5a-55) you can also specify the address of a dynamic area handler routine, which is called when the size of the area is being changed. The routine is called in SVC mode; the reason for calling it is given in a reason code held in R0. The section below gives the entry and exit conditions of the routine for each valid reason code.

When called, OS_ChangeDynamicArea is working. It rejects requests to resize dynamic areas. You should not use SWIs which resize dynamic areas, for example using OS_Module to claim some workspace. File operations should be normally avoided, although I/O on an existing file is usually safe.

PreGrow (0)

Issued just before pages are moved to grow an area

On entry

R0 = 0 (reason code)
R1 = pointer to a page block, the physical page numbers of which are set to –1; or undefined if bit 8 of the areas flags was clear on creation (see page 5a-55)
R2 = number of entries in page block (i.e. number of pages area is growing by)
R3 = amount area is growing by, in bytes (i.e. R2 × R5)
R4 = current size of area, in bytes
R5 = page size, in bytes
R12 = pointer to workspace

On exit

All registers preserved

Use

This reason code is issued when a call to OS_ChangeDynamicArea results in an area growing, before any pages are actually moved.

You can request that specific pages be used for growing the area by filling in their page numbers in the page block. If you do so, you must specify all the pages. The first entry in the page block corresponds to the lowest memory address of the extension, and the last entry in the page block the highest memory address.

You can prevent the area changing size by returning an error. R0 should point to a standard RISC OS error block, or be set to zero for a generic error message to be used. You should then return with the V flag set.

PostGrow (1)

Issued just after pages are moved to grow an area

On entry

R0 = 1 (reason code)
R1 = pointer to a page block, only the physical page numbers of which are defined; or undefined if bit 8 of the areas flags was clear on creation (see page 5a-55)
R2 = number of entries in page block (i.e. number of pages area grew by)
R3 = amount area grew by, in bytes (i.e. R2 × R5)
R4 = new size of area, in bytes
R5 = page size, in bytes
R12 = pointer to workspace

On exit

All registers preserved

Use

This reason code is issued when a call to OS_ChangeDynamicArea results in an area growing. It is called after the PreGrow reason code has been issued successfully and the memory pages have been moved. It provides the handler with a list of which physical pages have been moved into the area.

PreShrink (2)

Issued just before pages are moved to shrink an area

On entry

R0 = 2 (reason code)
R3 = amount area is shrinking by, in bytes
R4 = current size of area, in bytes
R5 = page size, in bytes
R12 = pointer to workspace

On exit

R3 = amount area can shrink by, in bytes (must be ≤ R3 on entry)
All other registers preserved

Use

This reason code is issued when a call to OS_ChangeDynamicArea results in an area shrinking, before any pages are moved. You can limit the amount of memory moved out of the area. If the permitted shrinkage you return is a non-page multiple, it will be rounded down to a page multiple.

You can prevent the area changing size by returning an error. R0 should point to a null terminated error message, or be set to zero for a generic error message to be used. R3 should be zero, to show that no shrinkage is possible. You should then return with the V flag set.

PostShrink (3)

Issued just after pages are moved to shrink an area

On entry

R0 = 3 (reason code)
R3 = amount area shrunk by
R4 = new size of area, in bytes
R5 = page size, in bytes
R12 = pointer to workspace

On exit

All registers preserved

Use

This reason code is issued when a call to OS_ChangeDynamicArea results in an area shrinking. It is called after the PreShrink reason code has been issued successfully even if the memory pages cannot be moved.

Sequence of actions when SWI OS_ChangeDynamicArea is called

The system stack is used for the page block passed to the PreGrow routine, where required. As a consequence there is a limit to the amount that an area can be grown by at one time. To get round this problem an area grow request of a large amount will be performed in several steps. If one of these steps fails then the grow will terminate early with the area grown by however much was achieved, but not by the full amount requested.

Two new service calls are used; Service_PagesUnsafe (page 5a-48) and Service_PagesSafe (page 5a-49). These are issued around page swapping to inform any DMA subsystems (eg IOMD DMA or second processor) that some pages are being swapped around.

Service calls

Service_PagesUnsafe (Service Call &8E)

Pages specified are about to be swapped for different pages

On entry

R1 = &8E (reason code)
R2 = page block filled in by the PreGrow routine, with the two address fields also filled in
R3 = number of pages in page block

On exit

All registers preserved

Use

This service call informs recipients that the pages specified are about to be swapped for different pages. Direct memory access activities involving the specified pages should be suspended until Service_PagesSafe (page 5a-49) has been received indicating the pages are safe.

You must not claim this service call.

This service call is only issued from RISC OS 3.5 onwards.

Service_PagesSafe (Service Call &8F)

Pages specified have been swapped for different pages

On entry

R1 = &8F (reason code)
R2 = number of entries in each page block
R3 = pointer to page block before move
R4 = pointer to page block after move

On exit

All registers preserved

Use

This service call informs recipients that the pages specified have been swapped for different pages and what those different pages are.

The logical addresses in both page blocks will match. The ‘before’ page block will contain the physical page numbers and physical addresses of the pages which were swapped, and the ‘after’ block the page numbers and physical addresses of the different pages which replaced them.

You must not claim this service call.

This service call is only issued from RISC OS 3.5 onwards.

Service_DynamicAreaCreate (Service Call &90)

Dynamic area has just been successfully created

On entry

R1 = &90 (reason code)
R2 = area number of dynamic area just created

On exit

All registers preserved

Use

This service call is issued just after the successful creation of a dynamic area.

This service call keeps the rest of the system informed about changes to the dynamic areas. It is used by the Task Manager, although other modules could make use of it.

You must not claim this service call.

This service call is only issued from RISC OS 3.5 onwards.

Service_DynamicAreaRemove (Service Call &91)

Dynamic area is about to be removed

On entry

R1 = &91 (reason code)
R2 = area number of dynamic area about to be removed

On exit

All registers preserved

Use

This service call is issued just before the removal of a dynamic area, after the area has been successfully reduced to zero size, but before it has been removed completely.

This service call keeps the rest of the system informed about changes to the dynamic areas. It is used by the Task Manager, although other modules could make use of it.

You must not claim this service call.

This service call is only issued from RISC OS 3.5 onwards.

Service_DynamicAreaRenumber (Service Call &92)

Dynamic area is being renumbered

On entry

R1 = &92 (reason code)
R2 = old area number
R3 = new area number

On exit

All registers preserved

Use

This service call is issued when a dynamic area is being renumbered.

This service call keeps the rest of the system informed about changes to the dynamic areas. It is used by the Task Manager, although other modules could make use of it.

You must not claim this service call.

This service call is only issued from RISC OS 3.5 onwards.

SWI calls

OS_DynamicArea (SWI &66)

Performs operations on dynamic areas

On entry

R0 = reason code
Other registers depend upon the reason code

On exit

R0 preserved
Other registers depend upon 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 perform operations on dynamic areas.

The particular action of OS_DynamicArea is given by the reason code in R0 as follows:

R0	Action	Page
0	Creates a new dynamic area	5a-55
1	Removes a previously created dynamic area	5a-58
2	Returns information on a dynamic area	5a-59
3	Enumerates dynamic areas	5a-60
4	Renumbers dynamic areas	5a-61

This call is only available from RISC OS 3.5 onwards.

Related SWIs

OS_ChangeDynamicArea (page 1-384)

Related vectors

None

OS_DynamicArea 0 (SWI &66)

Creates a new dynamic area

On entry

R0 = 0 (reason code)
R1 = –1 (or new area number not in range 128 - 255; this is reserved for Acorn use)
R2 = initial size of area, in bytes
R3 = –1 (or base logical address of area; this is reserved for Acorn use)
R4 = area flags (see below)
R5 = maximum size of area, in bytes (–1 ⟹ total RAM size of the machine)
R6 = pointer to dynamic area handler routine (see page 5a-45), or 0 if no routine
R7 = pointer to workspace, passed in R12 on entry to dynamic area handler routine;
or –1 for RISC OS to instead pass base address of area; or 0 if R6 = 0
R8 = pointer to null terminated string describing dynamic area (e.g. ‘Font cache’)

On exit

R0 preserved
R1 = allocated area number
R2 preserved
R3 = specified or allocated base address of area
R4 preserved
R5 = specified or allocated maximum size of area
R6 - R8 preserved

Use

This call creates a new dynamic area.

The area is created initially with size zero (no pages assigned to it), and is then grown to the size specified in R2, which involves calling the area handler (if any) pointed to by R6. The area’s maximum size is set to the lesser of the amount given in R5 on entry and the total RAM size of the machine; or to the total RAM size if R5 was –1 on entry.

RISC OS allocates a free area of logical address space which is big enough for the dynamic area’s maximum size. The base logical address is the lowest logical address used by that area. The area grows by adding pages at the high address end.

RISC OS allocates an area number itself, which is greater than or equal to 256. This means that a call to OS_ReadDynamicArea will always return the maximum area size in R2 for these areas.

The area flags passed in R4 are as follows:

Bit(s)	Meaning
0 - 3	access privileges to be given to each page in the area (same format as for OS_Read/SetMemMapEntries)
4	0 ⟹ area is bufferable 1 ⟹ area is not bufferable
5	0 ⟹ area is cacheable 1 ⟹ area is not cacheable
6	0 ⟹ area is singly mapped 1 ⟹ area is doubly mapped (reserved for Acorn use)
7	0 ⟹ area size may be dragged by the user in Task Manager window (has red bar) 1 ⟹ area size may not be dragged by the user in Task Manager window (has green bar)
8	0 ⟹ area does not require specific physical pages (ie R1 is undefined on entry to the PreGrow and PostGrow handlers) 1 ⟹ area may require specific physical pages (ie R1 points at a page block on entry to the PreGrow and PostGrow handlers)
9 - 31	reserved (must be zero)

The description string passed in R8 is used by the TaskManager in its display.

Once the area has been created, Service_DynamicAreaCreate is issued to inform the rest of the system about this change.

If the create dynamic area call returns an error for any reason, it may be assumed that the new area has not been created.

Notes for application writers

Applications should create only singly-mapped areas, and request that RISC OS allocate area numbers and logical addresses. This will prevent clashes of area numbers or addresses. For details of other usage, which has been provided largely for internal backwards compatibility, see the section entitled *System use* below.

Dynamic area handler routine

On entry, R6 points to the area handler routine which gets called with various reason codes when an area is grown or shrunk, and R7 specifies the workspace pointer that is passed to it in R12. If zero is passed in R6, then no routine will be called, and any shrink or grow will be allowed.

Details of the entry and exit conditions for this routine are given in the section entitled *Dynamic area handler routines* on page 5a-45

Errors

An error will be returned if:

- the given area number clashes with an existing area.
- the given base address is not on a memory page boundary.
- the logical address space occupied by the area at maximum size would intersect with any other area at maximum size.
- there is not enough contiguous logical address space to create the area.
- there is not enough memory in the free pool to allocate level 2 page tables to cover the area at maximum size.
- there is not enough memory to grow the area to the initial size requested.

System use

The following facilities are intended for internal system use only:

- The ability to create areas with specific area numbers.
- The ability to create areas at specific logical addresses.

  On entry, R3 holds the base address of the area, which must be aligned on a memory page boundary (to read the page size use OS_ReadMemMapInfo). With this usage, RISC OS does not allocate an area of logical address space for the dynamic area.
- The ability to create doubly-mapped areas.

  For doubly mapped areas the base logical address is the (fixed) boundary between the two mappings: the first mapping ends at R3 –1, and the second starts at R3. When one of these areas grows the pages in the first copy move down to accommodate the new pages at the end, and the second copy simply grows at the end.

OS_DynamicArea 1 (SWI &66)

Removes a previously created dynamic area

On entry

R0 = 1 (reason code)
R1 = area number

On exit

All registers preserved

Use

This call removes a previously created dynamic area.

Before the area is removed, RISC OS attempts to shrink it to zero size. This is done using OS_ChangeDynamicArea. If OS_ChangeDynamicArea returns an error, then the area will be grown back to its original size using OS_ChangeDynamicArea, and this call will return with an error. If OS_ChangeDynamicArea successfully reduced the area to zero size, then it will be removed.

Once the area has been removed Service_DynamicAreaRemove (page 5a-51) is issued to inform the rest of the system about this change.

An error is returned if the area was not removed for any reason.

OS_DynamicArea 2 (SWI &66)

Returns information on a dynamic area

On entry

R0 = 2 (reason code)
R1 = area number

On exit

R0, R1 preserved
R2 = current size of area, in bytes
R3 = base logical address of area
R4 = area flags
R5 = maximum size of area, in bytes
R6 = pointer to dynamic area handler routine (see page 5a-45), or 0 if no routine
R7 = pointer to workspace, passed in R12 on entry to dynamic area handler routine
R8 = pointer to null terminated string describing dynamic area (e.g. ‘Font cache’)

Use

This call returns information on a dynamic area.

For doubly-mapped areas, R3 on exit from this call returns the address of the boundary between the first and second copies of the area, whereas OS_ReadDynamicArea returns the start address of the first copy (for backwards compatibility).

OS_DynamicArea 3 (SWI &66)

Enumerates dynamic areas

On entry

R0 = 3 (reason code)
R1 = –1 to start enumeration, or area number

On exit

R1 = next area number, or –1 if no further areas

Use

This call enumerates dynamic areas.

This allows an application to find out what dynamic areas are defined. –1 is passed on entry to start the enumeration; the call is then repeated until –1 is returned on exit, which indicates that the enumeration has finished.

OS_DynamicArea 4 (SWI &66)

Renumbers dynamic areas

On entry

R0 = 4 (reason code)
R1 = old area number
R2 = new area number

On exit

R0 - R2 preserved

Use

This call renumbers dynamic areas, and is intended for system use only.

Once the area has been renumbered Service_DynamicAreaRenumber (page 5a-52) is issued to inform the rest of the system about this change.

An error is returned if the area specified by the old area number does not exist, or if the new number clashes with an existing area.

OS_Memory (SWI &68)

Performs miscellaneous operations for memory management

On entry

R0 = reason code (bits 0 - 7) and flags (bits 8 - 31, reason code specific)
Other registers depend upon the reason code

On exit

R0 preserved
Other registers depend upon 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 performs miscellaneous operations for memory management.

The particular action of OS_Memory is given by the reason code in bits 0 - 7 of R0 as follows:

R0	Action	page
0	General page block operations	5a-64
1 - 5	Reserved for system use	5a-66
6	Reads the size of the physical memory arrangement table	5a-67
7	Reads the physical memory arrangement table	5a-68
8	Reads the amount of a specified sort of memory available in the computer	5a-69
9	Reads controller presence and base address	5a-70

This call is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

OS_Memory 0 (SWI &68)

General page block operations

On entry

R0 = reason code and flags:

Bits	Meaning
0 - 7	0 (reason code)
8	physical page number provided when set
9	logical address provided when set
10	physical address provided when set
11	physical page number will be filled in when set (if bit 8 also clear)
12	logical address will be filled in when set (if bit 9 also clear)
13	physical address will be filled in when set (if bit 10 also clear)
14 - 15	cacheability control: 0 ⟹ no change 1 ⟹ no change 2 ⟹ disable cacheing on all specified pages 3 ⟹ enable cacheing on all specified pages
16 - 31	reserved (must be clear)

R1 = pointer to page block (see page 5a-42)
R2 = number of entries in page block

On exit

R0 - R2 preserved

Use

This call converts between the different memory spaces used to specify addresses in a page block: i.e. logical address, physical address, and physical page number. It can also alter the cacheability of pages. The addresses must be in RAM, but need not be page-aligned. You can do address conversions and control the cacheability on a per-page basis. You need not do any conversion when changing cacheability (i.e. bits 11 - 13 may be clear).

The page block is scanned and the specified operations applied to it. If any page is made physically uncacheable, then the cache will be flushed before the SWI exits. If any page cannot be converted or is non-existent then an error will be returned and the cacheability unaffected.

Cacheability is accumulated for each page. For example, if there are five clients which need cacheing turned off on a page, then each of them must turn cacheing back on individually for that page actually to become cached again.

Where an ambiguity may occur, for example in doubly-mapped areas such as the screen, one of the possible results will be chosen and filled in.

OS_Memory 1 - 5 (SWI &68)

These reason codes are for system use only; you must not use them in your own code.

OS_Memory 6 (SWI &68)

Reads the size of the physical memory arrangement table

On entry

R0 = 6 (reason code); all flags are reserved, so bits 8 - 31 must be clear

On exit

R0 preserved
R1 = table size (in bytes)
R2 = page size (in bytes)

Use

This call reads the size of the physical memory arrangement table, as returned by OS_Memory 7.

OS_Memory 7 (SWI &68)

Reads the physical memory arrangement table

On entry

R0 = 7 (reason code); all flags are reserved, so bits 8 - 31 must be clear
R1 = pointer to table to be filled in

On exit

R0, R1 preserved

Use

This call reads the physical memory arrangement table into the block of memory pointed to by R1. (You can find the required size of the block by calling OS_Memory 6.)

Each page of physical memory space has one entry in the table. Due to the large number of pages the table is packed down to only 4 bits per page. In each byte of the table the low order 4 bits correspond to the page before the high order 4 bits, i.e. the table is little-endian. This is the meaning of a nibble in the table:

Bit	Meaning
0 - 2	type of memory:
	0 not present
	1 DRAM
	2 VRAM
	3 ROM
	4 I/O
	5 - 7 undefined
3	0 ⇒ page available for allocation
	1 ⇒ page not available for allocation

The page availability is based on whether it is RAM, and whether it has already been allocated in such a way that it can't be replaced with a different RAM page eg the OS's page tables or screen memory.

If an area has particular requirements on the physical addresses used by it (eg if it needs contiguous physical memory for its area) we recommend that you issue this call inside the area's PreGrow handler, and then choose which pages to ask for on the basis of this information. This is preferable to issuing the call before you create the area, because the page availability may change during the process of creating the area.

OS_Memory 8 (SWI &68)

Reads the amount of a specified sort of memory available in the computer

On entry

R0 = reason code and flags:

Bits	Meaning
0 - 7	8 (reason code)
8 - 11	type of memory: 1 ⟹ DRAM 2 ⟹ VRAM 3 ⟹ ROM 4 ⟹ I/O
12 - 31	reserved (must be clear)

On exit

R0 preserved
R1 = number of pages of specified sort of memory
R2 = page size (in bytes)

Use

This call reads the amount of a specified sort of memory available in the computer. For I/O memory, all I/O memory is included: ie I/O space, VIDC space, and EASI space.

OS_Memory 9 (SWI &68)

Reads controller presence and base address

On entry

R0 = 9 (reason code); all flags are reserved, so bits 8 - 31 must be clear
R1 = controller ID:

Bit	Meaning
0 - 7	controller sequence number
8 - 31	controller type: 0 ⟹ EASI card access speed control (for internal use only) 1 ⟹ EASI space (for internal use only) 2 ⟹ VIDC1 3 ⟹ VIDC20

On exit

R0 preserved
R1 = controller base address, or 0 if not present.

Use

This call checks for the presence of a given controller, and returns its base address if it is fitted. Controllers are identified by type and sequence number so that a machine could be constructed with, say, more than one IDE controller in it.

For EASI space this call gives the base address of expansion card *n*, where *n* is the sequence number given. This reason code is provided for internal use only and is documented here for completeness' sake. In particular you must use the Expansion Card Manager to read this information and to control your expansion card's EASI space access speed.

OS_MMUControl (SWI &6B)

Modifies the ARM MMU control register

On entry

R0 = reason code and flags (must be zero)
R1 = XOR mask
R2 = AND mask

On exit

R1 = old value of control register
R2 = new value of control register

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call modifies the ARM MMU control register. The new value of the register is:

((old value AND R2) XOR R1)

The old value of the register is returned in R1, and the new value in R2. If the call results in the C (Cache enable) bit being changed, the cache is flushed.

This call is intended for internal system use only. Users wishing to enable or disable the cache should use the *Cache command instead.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

104 CMOS RAM allocation

Non-volatile memory

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 3.6 allocation

The full usage of CMOS RAM in RISC OS 3.6 is given below. Locations marked '†' were reserved for Acorn use in RISC OS 3.5, unless noted otherwise. Locations marked '‡' were not used, or were used for a different purpose under RISC OS 3.1. For details of CMOS RAM usage in RISC OS 3.1 and RISC OS 2, see page 1-361.

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 character of name – rest in bytes 158 - 172)
3	Econet printer server station id (0 ⇒ name configured)
4	Econet printer server net number (or first character of name – rest in bytes 153 - 157)
5	Default filing system number
6 - 7	*Unplug for ROM modules: 16 bits for up to 16 modules
8	Reserved for Acorn use
9 †	*Unplug for ROM modules: 8 bits for up to 8 modules
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	Misc configuration:	
	Bits 0 - 2 ‡	ADFS drive
	Bits 3 - 5	1 ⇒ ShCaps, 2 ⇒ NoCaps, 4 ⇒ Caps
	Bit 6	0 ⇒ Dir, 1 ⇒ NoDir
	Bit 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	WimpDoubleClickMove limit	
23	WimpAutoMenuDelay time	
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 no error
		Bit 1	DragASprite and DragAnObject: 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 †‡	type of click on toggle size icon that doesn't obscure icon bar: 0 ⇒ click, 1 ⇒ Shift-click
		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 ‡	Mouse type:		0 ⇒ standard quadrature mouse, 1 ⇒ Microsoft compatible serial mouse, 2 ⇒ Mouse Systems Corporation compatible serial mouse, 3 ⇒ PS/2 compatible serial mouse †, 4 - 255 reserved for Acorn use
30 - 45	**Reserved for the user**		
46 - 59	**Reserved for applications**		
60 - 79 ‡	Reserved for expansion card use		
80	Freeway net number in Acorn Access, or reserved for RISC *i*X		
81	&0F for Access to prevent RISC *i*X booting, or reserved for RISC *i*X		
82 - 105	ShareFS disc names in Acorn Access, or reserved for RISC *i*X		
106	ADFSFiler disc sharing for Acorn Access:		
		Bit 0	share drive 4 if set
		Bit 1	protect drive 4 if set
		Bit 2	share drive 5 if set
		Bit 3	protect drive 5 if set
		Bit 4	share drive 6 if set
		Bit 5	protect drive 6 if set
		Bit 6	share drive 7 if set
		Bit 7	protect drive 7 if set
	Or reserved for RISC *i*X		
107 - 111	Reserved for RISC *i*X		
112 - 127	Reserved for expansion card use		
128 - 129	Current year		
130 - 131 †	*Unplug for ROM modules: 16 bits for up to 16 modules		

132 DumpFormat, 16 bit sound control and quality:

- 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
- Bits 5 - 6 16 bit sound control:
  0 ⇒ standard μ-law sound (ie no 16 bit sound output)
  1 ⇒ DAC clock is slave, 11.2896MHz external clock, standard VIDC20 or 44.1kHz×4/(4…45) rates (as on ESP card)
  2 ⇒ DAC clock is slave, no external clock, standard VIDC20 rates only
  3 ⇒ DAC clock is master, external clock, suitable sound clock driver installed
- Bit 7 16 bit sound quality:
  0 ⇒ use specified sample rate
  1 ⇒ perform sample interpolation to keep sample rate over 25kHz

133 Sync, monitor type, some mode information:

- Bits 0, 7 0 ⇒ vertical sync, 1 ⇒ composite sync, 3 ⇒ auto sync)
- Bit 1 † 0 ⇒ enable LoadModeFile in !Boot,
  1 ⇒ disable LoadModeFile in !Boot
- Bits 2 - 6 monitor type: 0 ⇒ 0, 1 ⇒ 1, …, 31 ⇒ auto

134 FontSize in units of 4K

135 Number of ADFS drives:

- Bits 0 - 2 floppy disc drives
- Bits 3 - 5 ‡ no longer used (was ST506) – reserved for Acorn use
- Bits 6 - 7 IDE disc drives

136 ADFS floppy disc drive step rates:

- Bits 0 - 1 floppy disc drive 0
- Bits 2 - 3 floppy disc drive 1
- Bits 4 - 5 floppy disc drive 2
- Bits 6 - 7 floppy disc drive 3

137 ADFSbuffers

138	CDFS number of discs and buffer size:	
	Bits 0 - 4	number of CD-ROM drives
	Bits 5 - 7	buffer size: 0 ⇒ 0K, 1 ⇒ 8K, 2 ⇒ 16K, 3 ⇒ 32K, 4 ⇒ 64K, 5 ⇒ 128K, 6 ⇒ 256K, 7 ⇒ 512K
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 ⇒ 2D look, 1 ⇒ 3D look
	Bits 1 - 4 ‡	desktop font setting: 0 ⇒ use Wimp$Font… variables, 1 ⇒ use System font 2 - 15 ⇒ use font from ResourceFS
	Bits 5 - 6	reserved for Acorn use
	Bit 7 ‡	window background tiling: 0 ⇒ use tile_1, 1 ⇒ not tiled (i.e. grey 1)
141 - 142 †‡	*Unplug for ROM modules: 16 bits for up to 16 modules	
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, if hardware supports it
149 - 152	Allocated to BASIC Editor	
153 - 157	Printer server name characters 2 - 6 (character 1 at location 4)	
158 - 172	File server name characters 2 - 16 (character 1 at location 2)	
173 - 176	*Unplug for ROM modules: 32 bits for up to 32 modules	
177 - 184	*Unplug for expansion card modules: 8 × 8 bits for up to 8 modules per card	
185	Configured language	
186	Configured country	
187	*Unplug for network card modules: 8 bits for up to 8 modules	

188	Miscellaneous:	
	Bits 0 - 1	ROMFS Opt 4 state
	Bit 2	cache icon enable state: 0 ⇒ no cache icon state, 1 ⇒ caches icon
	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 for immediate Econet commands:	
	Bit 0	Peek
	Bit 1	Poke
	Bit 2	JSR
	Bit 3	User RPC
	Bit 4	OS RPC
	Bit 5	Halt
	Bit 6	GetRegs
	Bit 7	reserved for Acorn use
194	Mouse multiplier	
195	Miscellaneous:	
	Bit 0	AUN BootNet: 0 ⇒ disabled, 1 ⇒ enabled
	Bit 1	AUN dynamic station numbering: 0 ⇒ disabled, 1 ⇒ enabled
	Bit 2	type of last reset: 0 ⇒ ordinary, 1 ⇒ CMOS reset
	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 ARM: 0 ⇒ enabled, 1 ⇒ disabled
	Bit 6	broadcast loader enable: 0 ⇒ enabled, 1 ⇒ disabled
	Bit 7	colour hourglass enable: 0 ⇒ disabled, 1 ⇒ enabled
196	Mode and Wimp mode	
197	WimpFlags	

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 - 205	FontMax, FontMax1 - FontMax5	
206 - 207	Reserved for Acorn use	
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)	
215 - 216	Reserved for Acorn use	
217 - 219	*Unplug for ROM modules: 24 bits for up to 24 modules	
220	Alarm and time byte	
	Bits 0 - 2	format state: 0 ⇒ illegal (!Alarm checks for first run), 1 ⇒ analogue with seconds, 2 ⇒ analogue without seconds, 3 ⇒ HH:MM, 4 ⇒ format is '%24:%mi:%se', 5 ⇒ format is '%z12:%mi:%se %am %zd %zmn %yr', 6 & 7 reserved
	Bit 3	deletion: 0 ⇒ do not confirm, 1 ⇒ confirm
	Bit 4	auto save: 0 ⇒ no auto save, 1 ⇒ auto save
	Bit 5	5 day weeks: 0 ⇒ disabled, 1 ⇒ enabled
	Bit 6	alarm noise: 0 ⇒ not silent, 1 ⇒ silent
	Bit 7	Daylight Saving Time: 0 ⇒ normal time, 1 ⇒ Daylight Saving Time (DST)
221	WimpDragDelay time	
222	WimpDragMove limit	
223	WimpDoubleClickDelay time	
224 - 229	Local print server's name, stored by printer server software, or reserved for RISC *i*X	

230 †	LCD panel brightness and contrast, or reserved for RISC *i*X (was solely reserved for RISC *i*X under RISC OS 3.5)
231 †‡	*Unplug for ROM modules: 8 bits for up to 8 modules
232 ‡	Reserved for Acorn use
233 - 238 ‡	FSLock
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.

105 DMA

Introduction and Overview

From RISC OS 3.5 onwards, support for DMA (direct memory access) is provided by the new DMAManager module. However, some computers' hardware will not support DMA, in which case the DMA manager becomes dormant.

The DMA (Direct Memory Access) is controlled by four DMA channels; these service a potentially large number of devices.

DMA manager

The DMA manager:

- Performs the arbitration and switching between devices (with help from the device drivers).
- Provides a general purpose software interface to the DMA channels' available hardware interface.
- Isolates software from hardware so that changes to the hardware do not affect DMA clients – just the DMA manager.
- Handles memory mapping and memory management, so that any DMA clients are not concerned with logical to physical addresses or if a page is remapped during a DMA operation.

A DMA client registers itself with the DMA manager as the owner of a logical device. It then requests DMA transfers as and when necessary.

The DMA manager processes the requests on a first-come-first-served basis; it does not impose any priority on logical devices. It attempts to start the transfer as soon as possible. If the required DMA channel is not free, the request is stored in a FIFO queue. The request then starts when it is at the head of the queue and the required DMA channel is free.

The DMA manager provides a set of callback routines to keep the client up-to-date on the state of its operations; this is because of the possible time-lag between requesting and starting an operation.

DMA requests can be suspended and resumed, examined and terminated.

Technical details

Logical and physical DMA channels

The DMA manager controls the following physical DMA channels provided by IOMD:

0 General purpose channel 0
1 General purpose channel 1
2 General purpose channel 2
3 General purpose channel 3

The four general purpose physical channels must be shared by several devices via logical channels. The following logical channels are supported:

Logical channel	Use	Physical channel
&000	Expansion card 0, DMA line 0	2
&001	Expansion card 0, DMA line 1	
&010	Expansion card 1, DMA line 0	3
&011	Expansion card 1, DMA line 1	
&020	Expansion card 2, DMA line 0	
&021	Expansion card 2, DMA line 1	
&030	Expansion card 3, DMA line 0	
&031	Expansion card 3, DMA line 1	
&040	Expansion card 4, DMA line 0	
&041	Expansion card 4, DMA line 1	
&050	Expansion card 5, DMA line 0	
&051	Expansion card 5, DMA line 1	
&060	Expansion card 6, DMA line 0	
&061	Expansion card 6, DMA line 1	
&070	Expansion card 7, DMA line 0	
&071	Expansion card 7, DMA line 1	
&100	On-board SCSI	
&101	On-board Floppy	
&102	Parallel	
&103	Sound out	
&104	Sound in	
&105	Network card	0

Mapping between logical and physical channels

The mapping between logical and physical channels is fixed. Logical channels with no mapping shown above do not have DMA connected or are not controlled by the DMA Manager, and the numbers they have been assigned are for future use only.

The four general purpose physical DMA channels can be connected to devices on either side of the expansion card buffer. To avoid confusion, the expansion card buffer is not output enabled during DMA operations to internal peripherals, but is enabled for DMA operations to external devices. The DMA manager uses four bits in the IOMD register DMAEXT to specify whether the corresponding general purpose physical channel is mapped to an internal or external device.

Memory manager interfaces

The DMA manager and the memory manager interface in the following ways:

1. The DMA manager maps logical addresses to physical addresses so that the IOMD DMA registers can be programmed. It does so by creating a page table containing the logical addresses of all pages used in the transfer, and then calling OS_Memory (see page 5a-62) to get the memory manager to fill in the corresponding physical addresses.
2. On a DMA transfer from device to memory the DMA manager calls OS_Memory to ask the memory manager to mark the pages being DMAed into as uncacheable. This is so that reads from these pages return the transferred data rather than cached data. The DMA manager flushes the cache after making this call, but before starting the transfer. It makes the pages cacheable again once the transfer has completed.
3. The memory manager broadcasts Service_PagesUnsafe when it is about to remap some physical pages (i.e. when the physical addresses which correspond to a range of logical addresses are about to change). This service call provides a page table of the same form as that used in the OS_Memory interface which contains the physical addresses of the unsafe pages. The DMA manager then scans its page tables for all active transfers and temporarily halts any transfer which is transferring to or from an unsafe page. After the pages have been remapped the memory manager broadcasts Service_PagesSafe which provides the new physical addresses for the unsafe pages. The DMA manager then continues any halted transfers using the new physical addresses.

See the chapter entitled *Memory management* on page 5a-37 for more details.

Device drivers

Device drivers call DMA_RegisterChannel (page 5a-88) to register with the DMA manager which logical channels (devices) they control. The device drivers then call DMA_QueueTransfer (page 5a-91) to place DMA requests on a queue which the DMA manager processes in order. There are calls to terminate a transfer (DMA_TerminateTransfer – page 5a-93), suspend a transfer (DMA_SuspendTransfer – page 5a-95) and resume it (DMA_ResumeTransfer – page 5a-97), and to examine the state of a transfer (DMA_ExamineTransfer – page 5a-99). If a device wishes to relinquish control of a logical channel, it should do so by calling DMA_DeregisterChannel (page 5a-90).

Control routines

When a device driver calls DMA_RegisterChannel it passes a pointer to a word aligned table of control routine addresses. These control routines are called by the DMA manager during DMA; for a normal transfer the sequence is:

Start
Enable DMA
    transfer
    …
    transfer
Disable DMA
Completed

The control routines will be called in IRQ or SVC mode with interrupts enabled or disabled.

A control routine may alter processor mode as necessary. If interrupts are disabled on entry a control routine must neither enable interrupts, nor should it call DMA manager SWIs, as either may cause undesirable side effects. If interrupts are enabled on entry a control routine may change interrupt state and call DMA manager SWIs. On exit a control routine must restore the processor mode, interrupt status and processor flags (ie by using the instruction MOVS R15, R14 or an equivalent LDM), so that the DMA manager may continue where it left off. The only exception to this is that the Start control routine may alter the status of the V flag to indicate an error.

The control routines must conform to the following interfaces:

Enable DMA

On entry

R11 = R2 from DMA_QueueTransfer call
R12 = R5 from DMA_RegisterChannel call

On exit

All registers preserved

Use

The DMA manager calls this control routine to enable device DMA before starting the DMA transfers. It is assumed that the default state is for device DMA to be disabled.

Disable DMA

On entry

R11 = R2 from DMA_QueueTransfer call
R12 = R5 from DMA_RegisterChannel call

On exit

All registers preserved

Use

The DMA manager calls this control routine to disable device DMA. This may be called in mid transfer (for example, if DMA_TerminateTransfer or DMA_SuspendTransfer is called), or when a DMA request has completed.

Start

On entry

R11 = R2 from DMA_QueueTransfer call
R12 = R5 from DMA_RegisterChannel call

On exit

V set ⇒ R0 = pointer to error block
All other registers preserved

Use

The DMA manager calls this control routine before starting a new DMA request. This call is only made once for each DMA request; suspending and then resuming a transfer does not call this routine again. If the device driver no longer wants this operation to start then it should return with V set and R0 pointing to an error block; the Completed control routine is then called with the same error. Otherwise, the DMA manager then calls the Enable DMA control routine.

Completed

On entry

R0 = 0 (if V is clear) or pointer to error block (if V is set)
R11 = R2 from DMA_QueueTransfer call
R12 = R5 from DMA_RegisterChannel call

On exit

All registers preserved

Use

The DMA manager calls this control routine when a DMA request has completed. The Disable DMA control routine will have been called and the scatter list brought fully up to date before this routine is called. If the V flag is clear then the DMA request has completed successfully. Otherwise, the DMA request has terminated prematurely due to an error.

As soon as this control routine is called the DMA tag for the completed operation is no longer valid.

Possible errors include:

Error supplied to DMA_TerminateTransfer
Error returned from ‘Start’ control routine
‘DMA channel deregistered’

DMASync

On entry

R11 = R2 from DMA_QueueTransfer call
R12 = R5 from DMA_RegisterChannel call

On exit

R0 = 0 to continue, or *n* to stop after *n* bytes
All other registers preserved

Use

The DMA manager optionally calls this control routine after a fixed number of bytes have been transferred. The calling of this routine is configured when your device driver calls DMA_QueueTransfer (see page 5a-91) to queue a request for DMA transfer.

This routine allows for real-time synchronisation with DMA transfers, which is essential for time critical device drivers where the driver has to know how far a transfer has progressed.

If the device driver wants the transfer to stop then a non-zero value can be returned in R0 which specifies how many more bytes to transfer. Note that the DMA manager will attempt to stop after the specified number of bytes, but that this may not be possible because the next two sections of the transfer may have been initiated already. This means that the transfer might continue for at most

(2 × gap between DMASync calls + transfer unit size) bytes

If a number greater than or equal to this is returned by DMASync then the transfer is guaranteed to stop after the specified number of bytes.

SWI calls

DMA_RegisterChannel (SWI &46140)

Registers a client device as the controller of a logical channel

On entry

R0 = flags:
bits 0 - 31 reserved (must be set to 0)
R1 = logical channel
R2 = DMA cycle speed (0 - 3)
R3 = transfer unit size (1, 2, 4 or 16 bytes)
R4 = pointer to table of control routine addresses
R5 = workspace pointer to be passed to control routines in R12

On exit

R0 = channel registration handle
R1 - R5 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are not altered

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call registers a client device as the controller of a logical channel; it is typically called by a device driver. The value passed in R4 is a pointer to a word aligned table of control routine addresses:

Routine	Use
R4+0	Enable device DMA
R4+4	Disable device DMA
R4+8	Start
R4+12	Completed
R4+16	DMASync

These routines are called by the DMA manager to control the specified logical channel. They are called with R12 set to the value supplied in R5, which is usually the device driver's workspace pointer. For a full description of their use, see the section entitled *Control routines* on page 5a-84.

An error is returned if the logical channel is invalid or has already been claimed, an invalid cycle speed or transfer size is specified, or the control routine table is not word aligned.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

DMA_DeregisterChannel (page 5a-90)

Related vectors

None

DMA_DeregisterChannel (SWI &46141)

Deregisters a client device previously registered by DMA_RegisterChannel

On entry

R0 = channel registration handle

On exit

R0 preserved

Interrupts

Interrupts may be disabled
Fast interrupts are not altered

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call deregisters a client device previously registered with the DMA manager by DMA_RegisterChannel. Before the device is deregistered all DMA transfers will be terminated on the logical channel it is controlling.

An error is returned if the channel registration handle passed in R0 is invalid.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

DMA_RegisterChannel (page 5a-88)

Related vectors

None

DMA_QueueTransfer (SWI &46142)

Queues a DMA transfer request for a logical channel

On entry

R0 = flags:
bit 0 set ⇒ write (i.e. from memory to device),
clear ⇒ read (i.e. from device to memory)
bit 1 set ⇒ scatter list is a circular buffer, clear ⇒ not circular
bit 2 set ⇒ call DMASync control routine, clear ⇒ don't call
bits 3 - 31 reserved (must be set to 0)
R1 = channel registration handle
R2 = value of R11 to be passed to control routines
R3 = pointer to word-aligned scatter list
R4 = number of bytes to transfer, or 0 for infinite length transfer (if bit 1 of R0 set)
R5 = size of circular buffer (if bit 1 of R0 set)
R6 = number of bytes between calls to DMASync control routine (if bit 2 of R0 set)

On exit

R0 = DMA tag
All other registers preserved

Interrupts

Interrupts may be disabled
Fast interrupts are not altered

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call queues a DMA transfer request for a logical channel. The value in R2 is quoted in R11 when the DMA manager calls any of the control routines, and it describes the particular device/controller/transfer.

The scatter list is a word aligned table of (address, length) pairs, in that order:

- Both address and length are 32-bit values and are word aligned.
- The addresses are logical addresses which the client should not remap before the transfer is complete.
- The lengths are in bytes and are assumed to be a multiple of the transfer unit size specified when the logical device was registered.

When the transfer specified by a scatter list entry pair has completed the address is incremented and the length decremented to reflect how much data was transferred. The DMA manager then starts a transfer for the next pair and repeats until the total number of bytes specified in R4 have been transferred.

If bit 1 of R0 is set then the scatter list is treated as a circular buffer. This means that the scatter list will not be updated as described above, and will instead wrap at the end to start again at the beginning. In this case the transfer may be of infinite length, so R5 contains the size of the buffer. Transfers using circular buffers can be suspended and resumed, and can be terminated explicitly by calling DMA_TerminateTransfer (page 5a-93) or by the DMASync control routine (page 5a-87).

The value passed in R4 determines the number of bytes to be transferred, which must be a multiple of the transfer unit size. If the transfer uses a circular buffer then this value can be 0 to indicate an infinite length transfer. If the transfer doesn't use a circular buffer then this value must be less than or equal to the sum of the lengths of all scatter list entries.

If bit 2 of R0 is set then R6 contains the number of bytes which are to be transferred between successive calls to the device driver's DMASync control routine; again this value must be a multiple of the transfer unit size. This transfer method allows real-time synchronisation.

An error is returned if the channel registration handle is invalid, the scatter list is not word aligned, the length or the value in R5 or R6 (if either is used) is not a multiple of the transfer unit size, or the transfer is activated and the Start control routine returns an error.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

DMA_TerminateTransfer (page 5a-93)

Related vectors

None

DMA_TerminateTransfer (SWI &46143)

Terminates a DMA transfer

On entry

R0 = pointer to an error block
R1 = DMA tag

On exit

All registers preserved

Interrupts

Interrupts may be disabled
Fast interrupts are not altered

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call terminates a DMA transfer originally queued by DMA_QueueTransfer.

If the DMA transfer is active then it is stopped, and the DMA manager calls the Disable DMA control routine (page 5a-85); otherwise, the request is simply removed from its queue. The DMA manager then calls the Completed control routine (on page 5a-86) with V set and R0 pointing to the supplied error block.

If the terminated DMA transfer request was blocking a logical channel (i.e. had been suspended by a call to DMA_SuspendTransfer with bit 0 of R0 clear), then the logical channel is unblocked and its queued transfers are started again.

An error is returned if the DMA tag is invalid.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

DMA_QueueTransfer (page 5a-91)

Related vectors

None

DMA_SuspendTransfer (SWI &46144)

Suspends the given active DMA transfer

On entry

R0 = flags:
bit 0 clear ⟹ don't start queued transfers,
set ⟹ start next queued transfer
bits 1 - 31 reserved (must be set to 0)
R1 = DMA tag

On exit

All registers preserved

Interrupts

Interrupts may be disabled
Fast interrupts are not altered

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call suspends the given active DMA transfer. The DMA manager calls the Disable DMA control routine (page 5a-85), suspends the active DMA request, updates the scatter list and returns the request to a queue. If bit 0 of R0 is clear then no DMA requests for the same logical channel will be started until the suspended transfer is resumed or terminated.

An error is returned if the DMA tag is invalid, or the specified DMA transfer is not in progress.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

DMA_ResumeTransfer (page 5a-97)

Related vectors

None

DMA_ResumeTransfer (SWI &46145)

Resume a previously suspended DMA transfer

On entry

R0 = flags:
bits 0 - 31 reserved (must be set to 0)
R1 = DMA tag

On exit

All registers preserved

Interrupts

Interrupts may be disabled
Fast interrupts are not altered

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call resumes a previously suspended DMA transfer. A suspended transfer maintains its positions in the queue, so a resumed transfer has priority over requests queued after it was suspended. The DMA manager calls the Enable DMA control routine (page 5a-85) when the suspended transfer is restarted.

An error is returned if the DMA tag is invalid, or the buffer is not suspended.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

DMA_SuspendTransfer (page 5a-95)

Related vectors

None

DMA_ExamineTransfer (SWI &46146)

Returns the progress of a DMA transfer

On entry

R0 = flags:
bits 0 - 31 reserved (must be set to 0)
R1 = DMA tag

On exit

R0 = number of bytes transferred so far
All other registers preserved

Interrupts

Interrupt status may be disabled
Fast interrupts are not altered

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call returns progress of a DMA transfer, giving the total number of bytes transferred.

An error is returned if the DMA tag is invalid.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

106 Video

Introduction

The new architecture uses a new video controller – VIDC20. Using this gives a much improved video capability over the previous generation of computers that used VIDC1 or VIDC1a chips. In particular, it supports pixel depths of up to 32*bpp* (bits per pixel) on a much wider range of monitor types.

The video system was substantially changed in RISC OS 3.5, so that VIDC20 can be used to its full capabilities. There are new ways of selecting and specifying screen modes and monitor types. The sprite format has been extended to allow sprites that use the new screen modes. Many calls have been extended to support these new features, particularly in the kernel (including OS_SpriteOp calls) and in ColourTrans.

In RISC OS 3.6, further minor extensions have been made to the video system: in particular, support has been added for palettes in the new sprite format.

Full details are in the rest of this chapter. **The information in this chapter applies from RISC OS 3.5 onwards**, unless otherwise stated.

Furthermore, support for JPEG files has been added in RISC OS 3.6. This is described separately, in the chapter *JPEG images* on page 5a-145.

Overview

New ways of selecting modes

There are new ways of specifying and selecting screen modes, to take account of the much wider range of screen modes potentially available. See *Mode selection* on page 5a-104.

Many calls have been extended to use these new methods; see their individual entries in the section entitled *Technical details* on page 5a-104. A few calls have been added where existing calls could not be extended, or to provide functionality not present in RISC OS 3.

Monitor configuration stored in files

The new architecture can support a much wider range of monitors than in the past.

At start-up the monitor type, screen mode and sync are set from configured values, as in RISC OS 3. However, the new architecture only detects bit 0 of the monitor lead ID, so auto-configuration of monitor type, screen mode and sync is no longer so flexible (see *Service_MonitorLeadTranslation* on page 5a-114).

These defaults are then overridden by information read from a monitor-specific file, each of which stores the full range of modes available for a specific monitor. For full details, see *ModeInfo files* on page 5a-106, and *The ScreenModes module* on page 5a-108.

New sprite format

The RISC OS 3 sprite format has been extended to support the new pixel depths that are available. The new format is defined on page 5a-110.

OS_SpriteOp calls have also been extended to work with the new sprite format; see *Changes to OS_SpriteOp* on page 5a-117. Calls that create sprites will – where possible – create an old format sprite. This is to ease exchange of files with machines running older versions of RISC OS.

ColourTrans extended to support new modes and sprites

ColourTrans calls have been extended to support the changes to other parts of the video software. For full details see *Changes to existing ColourTrans SWIs* on page 5a-120.

New PaletteV reason codes

The PaletteV software vector has been enhanced by the addition of extra reason codes. These are for block reading and writing of the palette, and setting up gamma correction tables for RGB values being programmed into the palette. See page 5a-113, and the description of the new reason codes on page 5a-123.

All 8 bits of colour numbers are significant

All 8 bits of a colour component are now significant. Do not work in four bit quantities and fill in the lower nibble either by setting it to zero or by copying the upper nibble. This technique still works, but only allows access to sixteen of the possible 256 intensities.

Technical details

Mode selection

Because of the increased number of colours and the range of resolutions available, using a mode number to define screen modes has become limiting. (There was a maximum of only 128 modes, with just 64 available to Acorn.) To bypass this limitation a number of different methods are now used to define modes. These methods and their terminology are summarised below.

Mode numbers

The *mode numbers* used in earlier versions of RISC OS are still supported.

Mode selectors

A *mode selector* is a word-aligned structure that defines a particular mode. This includes its resolution, numbers of colours, frame rate and other variables.

A mode selector has the following format:

Offset	Value
0	mode selector flags: bit 0 = 1 (see *Distinguishing mode selectors from sprite areas* on page 5a-108) bits 1 to 7 = format specifier (zero for this format) bits 8 to 31 = other flags (reserved - must be zero)
4	x-resolution (in pixels)
8	y-resolution (in pixels)
12	pixel depth: 0 ⇒ 1bpp, 1 ⇒ 2bpp, 2 ⇒ 4bpp, 3 ⇒ 8bpp, 4 ⇒ 16bpp, 5 ⇒ 32bpp
16	frame rate (in Hz); –1 ⇒ use highest rate available
20	pairs of [mode variable index, value] words; there may be any number of these, including zero
n	–1 (terminator)

The mode variable indexes mentioned here are the same numbers which specify mode variables in the SWI OS_ReadModeVariable. See page 1-736 for more information.

Mode specifiers

A *mode specifier* is a word passed to a SWI to specify a mode. A mode specifier may be either a mode number (0 - 255), or a pointer to a mode selector (greater than 255). The range of the value determines which it is.

Mode strings

Mode strings are a textual form of mode selection, used by several Wimp calls and the command *WimpMode, as well as the Display manager utility.

A mode string has the following syntax:

Syntax	**Meaning**
X*nnnn*	X resolution (*nnnn* is three or four digits)
Y*nnnn*	Y resolution (*nnnn* is three or four digits)
C*ccc*	Colours (*ccc* = 2, 4, 16, 64, 256, 32T, 32K, 16M)
G*ggg*	Greys (*ggg* = 4, 16, 256)
EX*n*	X EIG factor (*n* = 0 to 3, smaller values make text larger)
EY*n*	Y EIG factor (*n* = 0 to 3, smaller values make text larger)
F*fff*	Frame rate (Hz) (*fff* is two or three digits)

For example:

X640 Y512 C16	Mode 20 (nb not supported by all monitors)
X640 Y480 C16 EX0 EY0	Mode 27 with extra-large text
X320,Y480,C64	VIDC 1 style 8bpp, VGA with rectangular pixels

- The parameters G and C cannot be specified together.
- Parameters EX and EY are optional and the default size is used if they are not given.
- Parameter F is optional; if it is omitted a value of –1 is used, which uses the highest frame rate available.

The Display manager utility only changes modes using *WimpMode. A mode selection string is constructed when the user clicks on OK in the window. If you want to use a mode number, you can enter it instead of a mode string. For example you can enter 15 to select the old mode 15.

ModeInfo files

ModeInfo files contain definitions of all the screen modes available on a particular monitor. The mode definitions are written in plain text, so the files can be edited.

- Spaces and tab characters (&09) are allowed anywhere in the file except in the middle of keywords or numbers.
- Lines starting with any number of spaces or tabs followed by the hash character '#' are treated as comments and ignored.

Header

The file always starts with the following two lines:

```
file_format: format
monitor_title: title
```

where:

format	must be 1 for this format file
title	is a textual description of this type of monitor

This may be followed by an optional line to control DPMS power saving (see *Monitor power saving* on page 5a-653):

```
DPMS_state: state
```

where:

state	must be 0, 1, 2 or 3

Under RISC OS 3.6 or later, the first two lines may instead be followed by an optional line to indicate the file is for an LCD panel:

```
lcd_support: value
```

where:

value	must be 1 (single panel) or 2 (dual panel)

Mode definitions

The header is followed by any number of mode definitions, as follows:

```
startmode
        mode_name: mode_name
        x_res: x-resolution
        y_res: y-resolution
        h_timings: hsync, hbpch, hlbdr, hdisp, hrbdr, hfpch
        v_timings: vsync, vbpch, vtbdr, vdisp, vbbdr, vfpch
        pixel_rate: pixel_rate
        sync_pol: sync_polarities
endmode
```

where:

mode_name	is a textual name for the mode for use in menus and such. The mode name field must be present; however, the *mode name* itself may be blank, which prevents the mode appearing on menus (eg the Display Manager's Resolution menu.)
x-resolution	is the number of pixels displayed across the screen
y-resolution	is the number of displayed rasters
hsync	is the width of the hsync pulse
hbpch	is the width of the horizontal back porch
hlbdr	is the width of the left hand border
hdisp	is the number of displayed pixels horizontally, which is normally the same as *x-resolution*)
hrbdr	is the width of the right hand border
hfpch	is the width of the horizontal front porch
vsync	is the width of the vsync pulse
vbpch	is the width of the vertical back porch
vtbdr	is the width of the top border
vdisp	is the number of displayed rasters vertically (normally the same as *y-resolution*)
vbbdr	is the width of the bottom border
vfpch	is the width of the vertical front porch
pixel_rate	is the pixel rate in kHz
sync_polarities	is a number indicating what kind of sync signals are required, as follows: 0 hsync normal, vsync normal 1 hsync inverted, vsync normal 2 hsync normal, vsync inverted 3 hsync inverted, vsync inverted

All values on the h_timings line are in units of pixels, and all values on the v_timings line are in units of raster lines.

Note: VIDC20 imposes restrictions on these parameters. In particular, all the horizontal timing values must be multiples of 2, and the horizontal total (hsync + hbpch + hlbdr + hdisp + hrbdr + hfpch) must be a multiple of 4. See the VIDC20 data sheet for details of further restrictions.

ModeInfo files are used by the new ScreenModes module; see *The ScreenModes module* on page 5a-108.

Distinguishing mode selectors from sprite areas

In earlier versions of RISC OS, certain SWI calls (e.g. ColourTrans ones) were passed screen modes either as mode numbers, or as pointers to sprite areas from which the sprite's screen mode was read. The two were differentiated by their values.

These calls now also accept a pointer to a mode selector, which must be distinguished from a pointer to a sprite area. This is done by examining the first word of the area pointed to. The first word of a sprite area is the size of the area; since the area must be word aligned, bit 0 of this word will always be 0.For this reason a mode selector always has bit 0 of its first word set, thus ensuring it can be distinguished from a sprite area.

The ScreenModes module

VIDC20 supports a much wider range of monitors than did VIDC1, with a wide range of available line frequencies. Supporting these requires a new mechanism, without which RISC OS would have had to define many new monitor type numbers.

Monitors are now supported using *ModeInfo* files (page 5a-106) held on hard disc. Each file holds the timings for a full set of screen modes on a particular monitor. A new ScreenModes module provides a * Command – *LoadModeFile (page 5a-142) – to load one of these files. If the file contains valid information, the ScreenModes module then calls OS_ScreenMode 3 (page 5a-135) to set the current MonitorType to 7 (file).

This makes available all the screen modes defined in the file, while removing all modes defined in any previously loaded file.

Desktop screen modes

Colours

The following colour options are supported on the desktop:

- 1 bpp (monochrome)
- 2 bpp (grey)
- 4 bpp (grey)
- 4bpp (colour)
- 8 bpp (palette set to correspond with default VIDC1 operation – using tints)
- 8 bpp (palette set to provide 256 grey levels)
- 16 bpp (palette fixed, can only be used for Gamma correction)
- 32 bpp (palette fixed, can only be used for Gamma correction).

This table shows how the bits per pixel value corresponds to the number of colours available:

Bits per pixel	Number of colours
1	2
2	4
4	16
8	256 (or 64)
16	32 thousand
32	16 million

Screen memory

The limits of the capabilities of the VIDC20 depend upon the amount of screen memory available. The new architecture can use either DRAM or VRAM based screen memory.

- VRAM based screen memory can be 1MB or 2MB in size.
- DRAM based screen memory is limited to 1MB. There is the same trade-off as in RISC OS 3 between the bandwidth used for video and that used for other purposes, such as running applications.

Screen resolutions

The following are the maximum screen resolutions that the desktop supports. These figures are maximum limits, and are for guidance only; some monitors will not be able to display all these resolutions. Your software should not assume that any particular combination is possible.

Using DRAM as screen memory

1024 × 768	4 bpp
800 × 600	8 bpp
768 × 288, or 480 × 352	16 bpp
384 × 288	32 bpp

Using 1MB of VRAM as screen memory

1280 × 1024	4 bpp
1024 × 768	8 bpp
800 × 600	16 bpp
768 × 288, or 480 × 352	32 bpp

Using 2MB of VRAM as screen memory

1280 × 1024	4 bpp
1280 × 1024	8 bpp
1024 × 768	16 bpp
800 × 600	32 bpp

The Display manager utility allows a selection of pre-defined modes to be chosen; custom modes can also be used and defined.

Grey level modes and the Wimp

The Window Manager now allows the selection of 16 and 256 level grey scale modes in the desktop.

In 16 grey-level modes the first eight desktop ‘colours’ are the same shades of grey as in normal 16 colour modes, and the next eight ‘colours’ provide interpolated greys. This means that the logical colours do not decrease in brightness monotonically.

In 256 grey-level modes the palette is set up so that a pixel value of 0 is black, 255 is white, and the values between form a linear grey scale.

New format of a sprite

The sprite format has been extended to incorporate the new modes. The new format avoids the problems that could be caused in previous versions of RISC OS by binding sprite files to a mode number not available on the viewing computer. The new format uses different *sprite types* for different pixel depths.

The old sprite format (page 1-777) is still fully supported. RISC OS distinguishes between new and old format sprites by examining the top bits of the word that specified the sprite mode in the old format. These bits were always zero for the old format; in the new format these are used to store the (non-zero) sprite type.

Sprite Control Block

The Sprite Control Block has this format **for new sprite types**:

Bytes	**Meaning**
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 *	0 (reserved for future use) – no left hand wastage is allowed on new format sprites

Bytes	Meaning
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 *	New *sprite mode word*:

Bits	Meaning
27 - 31	Sprite type: 0 ⇒ old type; see page 1-777 for format 1 - 31 ⇒ new type; these are defined below
14 - 26	Vertical dpi; should be 180, 90, 45 or 22
1 - 13	Horizontal dpi; should be 180, 90, 45 or 22
0	1; see *Distinguishing sprite modes and mode selectors* on page 5a-112

44…	Palette data (optional)

* These words have changed from the old format of sprite control block.

Sprite types

Sprite types are as follows.

Type	Meaning
0	Backward compatible mode – see page 1-777
1	1bpp image; 1bpp mask; palette not supported by RISC OS 3.5
2	2bpp image; 1bpp mask; palette not supported by RISC OS 3.5
3	4bpp image; 1bpp mask; palette not supported by RISC OS 3.5
4	8bpp image; 1bpp mask; palette not supported by RISC OS 3.5
5	16bpp image; 1bpp mask; palette not supported by RISC OS
6	32bpp image; 1bpp mask; palette not supported by RISC OS
7	Allocated for CMYK, but not supported within RISC OS
8	Allocated for 24bpp, but not supported within RISC OS
9 - 31	Reserved for future expansion

Pixel formats for 16 and 32bpp sprites

The formats of pixels in 16 and 32bpp sprites is as follows:

16bpp sprites

Bit	Use
0 - 4	Red
5 - 9	Green
10 - 14	Blue
15	Reserved (set to 0)

32bpp sprites

Bit	Use
0 - 7	Red
8 - 15	Green
16 - 23	Blue
24 - 31	Reserved (set to 0)

Mask data structure

Whatever the depth of image, the mask for new type sprites is 1 bit per pixel. Each row of mask bits begins word aligned. The layout of mask bits is identical to the layout of a 1bpp sprite's image data.

Sprite palettes

RISC OS 3.5 does not support palettes for new type sprites, and will generate an error if it finds one.

RISC OS 3.6 adds support for palettes in new type sprites that have up to 8bpp. It does not support palettes in 16 or 32bpp sprites. The format of the palette data is the same as for old type sprites.

Distinguishing sprite modes and mode selectors

You can now specify modes in some calls using either a mode specifier (i.e. a mode number, as before, or a pointer to a mode selector), or a new sprite mode word. The passed value is treated as a mode number if it is less than 256. The other two cases must be distinguished somehow. It is for this reason that bit 0 is set for a new sprite mode word; this bit is always clear in a pointer to a mode selector, since they must be word-aligned.

New software vectors

PaletteV has been enhanced by the addition of extra reason codes. For a full description see page 5a-123.

Block read and write

There are new reason codes for block reading and writing of the palette. A number of other calls have been extended to use these reason codes in preference, before falling back to older methods of reading the palette should this fail.

Gamma correction

A further reason code can be used to set up gamma correction tables for RGB values being programmed into the palette.

Claiming PaletteV

Not all PaletteV claimants support this code, so care must be take in the use of these calls. The correct behaviour for a claimant is to return all calls, but only set R4 to 0 for those it knows. This avoids problems with different PaletteV claimants processing some reason codes and passing on others it does not understand.

New Service Call

A new service call has been added:

- Service_EnumerateScreenModes (page 5a-129) enumerates the available screen modes. Applications should not issue this service call themselves, but should instead use the front-end provided by the new SWI OS_ScreenMode 2 (page 5a-135).

Changes to existing Service Calls

Service_ModeExtension (page 1-641)

Service_ModeExtension now uses a new format of VIDC list that is independent of the video controller used. There is no support for the old formats of VIDC list used in earlier versions of RISC OS, which included values corresponding directly to VIDC1 register formats. In particular, this means that old mode extend modules **will not work** under RISC OS 3.5 or later. For more information see page 5a-125.

Service_ModeTranslation (page 1-645)

This service call has been extended to allow the substitute mode passed back in R2 to be an arbitrary mode specifier. However, the input mode will only ever be a mode number, as a mode change controlled by a pointer to a mode selector never uses a substitute mode.

Service_MonitorLeadTranslation (page 1-646)

The new architecture can only detect the state of the ID0 pin, and so the defaults set by this service call have been changed. New defaults are:

ID0	Monitor type	Sync type	Default mode
H	0 (TV standard)	1 (composite)	12
1	0 (TV standard)	1 (composite)	12
0	3 (VGA)	0 (separate)	27

Mono VGA monitors are interpreted as TV standard monitors, so this class of monitor requires manual configuration before use. Other monitor types are detected and an appropriate mode is selected.

Changes to existing VDU calls

VDU 17 (page 1-585) and VDU 18 (page 1-586)

These calls have not been extended to work in 16bpp or 32bpp modes; in such cases they will behave as if in an 8bpp mode. This makes the calls no longer useful, and you should instead use OS_SetColour (page 1-754 and page 5a-116). The colour number to be used can be found by using ColourTrans_ReturnColourNumber.

VDU 19 (page 1-588)

In 16bpp and 32bpp modes, the palette is altered for gamma matching only. VDU 19 should not be used in these modes. It is no longer necessary to duplicate nibbles – all 8 bits of the colour component are significant.

VDU 22 (page 1-594)

This call has not been altered, and so no longer allows all display modes to be chosen. You should no longer use VDU 22; for full access to the screen modes available on the computer you should instead use OS_ScreenMode (page 5a-133).

VDU 23,17,0-3 (page 1-616)

This call works for all 8bpp screen modes, but is of little use for modes having a full 256 colour palette.

VDU 25,144-167 (page 1-628)

These calls to plot circles do not work properly with 180 × 45 or 45 × 180 screen modes.

New kernel SWI

The following new kernel SWI has been created. It is defined in full at the end of this chapter:

- OS_ScreenMode (page 5a-133) performs miscellaneous operations for screen mode handling. R0 provides a reason code which determines which operation is performed.

Changes to existing kernel SWIs

OS_Byte 135 (page 1-670)

The returned mode may now be a mode specifier.

OS_Word 9 (page 1-700)

You should no longer use OS_Word 9 to read the pixel logical colour.

OS_Word 11 (page 1-704)

Use ColourTrans_ReadPalette (page 3-393) in preference to this call.

OS_Word 12 (page 1-706)

Use ColourTrans_WritePalette (page 3-395) in preference to this call.

OS_ReadModeVariable (page 1-736)

You can now specify the mode using a mode specifier or a new sprite mode word.

A new ModeFlag has been assigned; bit 7 is set to show that an 8bpp mode uses a full palette. Such modes also return 255 for NColour, whereas old-style 8bpp modes still return 63.

If you are using a new sprite mode word, the values returned for certain VDU variables depends on its sprite type ('T' below), and its horizontal and vertical dpi ('hdpi' and 'vdpi' respectively):

Name	No.	Returned value
NColour	3	T=1 ⇒ 1, T=2 ⇒ 3, T=3 ⇒ 15, T=4 ⇒ 63 or 255, T=5 ⇒ 65535, T=6 ⇒ &FFFFFFFF
XEigFactor	4	hdpi=22/23 ⇒ 3, hdpi=45 ⇒ 2, hdpi=90 ⇒ 1, hdpi=180 ⇒ 0
YEigFactor	5	vdpi=22/23 ⇒ 3, vdpi=45 ⇒ 2, vdpi=90 ⇒ 1, vdpi=180 ⇒ 0
Log2BPP	9	T=1 ⇒ 0, T=2 ⇒ 1, T=3 ⇒ 2, T=4 ⇒ 3, T=5 ⇒ 4, T=6 ⇒ 5
Log2BPC	10	T=1 ⇒ 0, T=2 ⇒ 1, T=3 ⇒ 2, T=4 ⇒ 3, T=5 ⇒ 4, T=6 ⇒ 5

OS_CheckModeValid (page 1-742)

This call now accepts a mode specifier in R0, not just a mode number. In addition, the returned substitute mode may be a mode specifier.

OS_Plot (page 1-744)

See VDU 25,144-167 (page 5a-115).

OS_SetColour (page 1-754)

This call now has two extra flag bits defined in R0. You can now read or write the text and graphics foreground/background colours. The new flags are:

bit 6	set ⇒ R1 = text colour, clear ⇒ R1 = graphics colour
bit 7	set ⇒ read colour, clear ⇒ set colour

When setting the colour all the flags are used. As before, you can specify graphics colours using a pattern block or a colour number; text colours must be specified as colour numbers.

When reading the colour only the foreground/background flag and the text/graphics flag are used. For graphics colours you **must** supply a pattern block, whereas text colours are returned in R1 as colour numbers.

The values returned can be passed straight back to OS_SetColour to restore the colour.

New OS_SpriteOp reason code

The following new reason code has been added to OS_SpriteOp. It is defined in full at the end of this chapter:

- OS_SpriteOp 17 (page 5a-131) checks the validity of a sprite area.

Changes to OS_SpriteOp

OS_SpriteOp calls support new type sprites wherever appropriate. This is assumed, and **not** stated below for each call.

Support for particular features has only been added as they have become legal in RISC OS. In particular:

- Since RISC OS 3.5 did not allow new type sprites with a palette, you cannot use its OS_SpriteOp calls to create them. RISC OS 3.6 allows new type sprites with up to 8bpp to have a palette, and so you can use the relevant calls to create such sprites.
- However, even though new type sprites with a mask were legal under RISC OS 3.5, a few of its calls did not accept such sprites. These calls are specifically noted below.

Forcing a mode number when creating sprites

Calls that create sprites will – where possible – create an old format sprite. This is to ease exchange of files with machines running older versions of RISC OS. In doing so, they may have to force the current mode from one specified by a mode selector back to a mode number. The following modes are used:

	1bpp	**2bpp**	**4bpp**	**8bpp**
90 × 45 dpi	0	8	12	15
45 × 45 dpi	—	1	9	13
90 × 90 dpi	25	26	27	28

Combinations not shown in the above table (including 45×45dpi at 1bpp) cannot be forced back to a mode number, and must always use the new sprite types.

Use of translation tables and palette entries

In general you must supply a translation table when plotting a sprite with 8bpp or less to a 16 or 32 bpp mode. However, a few calls (noted in their descriptions below) provide a new flag to force sprites to be plotted using their palette entries rather than translation tables. The sprites must have full palette entries for you to do this.

Under RISC OS 3.5 only, when plotting an 8bpp sprite with a full palette to a 16 or 32bpp mode, it is plotted from its palette entries. However, for compatibility we suggest that you do not rely on this, but instead use the new flag where relevant.

OS_SpriteOp 15 – Create sprite (page 1-800)

On entry R6 can now be a mode number (as before), or a sprite mode word, or a pointer to a mode selector.

OS_SpriteOp 31 – Insert row (page 1-809)
OS_SpriteOp 32 – Delete row (page 1-810)
OS_SpriteOp 33 – Flip about x axis (page 1-811)
OS_SpriteOp 35 – Append sprite (page 1-813)

Under RISC OS 3.5 these calls do not accept new type sprites that have a mask. This restriction has been removed from RISC OS 3.6 onwards.

Note that OS_SpriteOp 35 will generate an error if you attempt to append a sprite with a 1bpp mask to one with a non-1bpp mask.

OS_SpriteOp 36 – Set pointer shape (page 1-815)

This call accepts type 0 - 4 sprites. It does not accept types 5 and 6.

OS_SpriteOp 37 – Create/remove palette (page 1-817)

Under RISC OS 3.5 this call accepts new type sprites. It generates an error if you try to create a palette, but does nothing if you try to remove one; this is because RISC OS 3.5 does not support new type sprite palettes.

RISC OS 3.6 allows new type sprites with up to 8bpp to have a palette. This call therefore can create or remove palettes from such sprites. New type sprites of a greater depth (ie > 8bpp) are still accepted, but treated just as in RISC OS 3.5, since such sprites are still not allowed to have a palette.

OS_SpriteOp 45 – Insert column (page 1-823)
OS_SpriteOp 46 – Delete column (page 1-824)
OS_SpriteOp 47 – Flip about y axis (page 1-825)

Under RISC OS 3.5 these calls do not accept new type sprites that have a mask. This restriction has been removed from RISC OS 3.6 onwards.

OS_SpriteOp 52 – Put sprite scaled (page 1-830)

Bit 4 of R5, if set, forces a 1, 2 or 4bpp sprite to be plotted into 16 or 32bpp using its palette entries rather than a translation table. The sprite must have a full palette.

From RISC OS 3.6 onwards two further flags have been added:

- Bit 5 of R5, when set, indicates that a *wide translation table* is being used. The table is 1 byte wide when plotting into less than 8bpp, 2 bytes wide for 16bpp, and 4 bytes wide for 32bpp. This is similar to the action of bit 4 of R5 in ColourTrans_SelectTable (page 3-344).
- Bit 6 of R5, when set, makes RISC OS use dithering when plotting a 16 or 32bpp sprite to a reduced depth. This bit is otherwise ignored.

OS_SpriteOp 53 – Put sprite greyscaled (page 1-831)

From RISC OS 3.6 onwards this call is no longer available. If you attempt to call it, the Sprite Extend module generates an appropriate error.

OS_SpriteOp 54 – Remove lefthand wastage (page 1-832)

This call will not accept new type sprites that have a mask. However, you should never need to call it for new type sprites, since they are not allowed to have any lefthand wastage.

OS_SpriteOp 56 – Put sprite transformed (page 1-833)

Bit 4 of R5, if set, forces a 1, 2 or 4bpp sprite to be plotted into 16 or 32bpp using its palette entries rather than a translation table. The sprite must have a full palette.

From RISC OS 3.6 onwards, bit 5 of R5, when set, indicates that a *wide translation table* is being used. The table is 1 byte wide when plotting into less than 8bpp, 2 bytes wide for 16bpp, and 4 bytes wide for 32bpp. This is similar to the action of bit 4 of R5 in ColourTrans_SelectTable (page 3-344).

OS_SpriteOp 57 and 58 – Insert/delete rows/columns (page 1-836)

Under RISC OS 3.5 these calls do not accept new type sprites that have a mask. This restriction has been removed from RISC OS 3.6 onwards.

Changes to existing Wimp SWIs

Wimp_SetMode (page 3-185)

This call now accepts a mode specifier in R0. If this is a pointer to a mode selector, the Wimp copies the mode selector, so you can then re-use the memory.

Changes to existing ColourTrans SWIs

ColourTrans has been extended to support the new 16bpp and 32bpp modes. Facilities have been provided to allow behaviour in these depths to be backwards compatible.

All ColourTrans calls will now accept a mode specifier rather than a mode number, where appropriate. Most ColourTrans calls that use GCOLs will use 16 or 32 bit values for 16 or 32bpp modes. Only the exceptions are noted below.

ColourTrans_SelectTable (page 3-344)
ColourTrans_GenerateTable (page 3-405)

The table size generated by an application attempting to map down from a 16 or 32bpp to a 1-8bpp mode would be excessively large, so ColourTrans does not return full translation tables in these cases. There is no longer a relationship between the size of the table returned by ColourTrans and the number of colours in the source mode. You must determine the size of the table before requesting it.

The revised translation table functionality is:

		Source Mode	
		1, 2, 4, 8 bpp	**16, 32 bpp**
Destination Mode	**1, 2, 4, 8, bpp**	See note 1 below	See note 2 below2
	16, 32 bpp	See note 3 below	See note 4 below

1 This is the existing RISC OS 3.1 algorithm unchanged.

2 This returns a structure including a pointer to a 32 KB table mapping from 5 bits per primary colour to a colour number in the destination screen mode.

The structure of the table is:

0 Word = &2E4B3233 (‘32K.’)
4 Pointer to table
8 Word = &2E4B3233 (‘32K.’)

The guard words each side of the pointer allow SpriteExtend to check whether the translation table passed to it is of this form, or is a direct look up table.

The most commonly used tables are precalculated. In other cases the table must be calculated when it is first requested, which may take a few seconds. The table remains valid until the next palette change, mode change or switch output to

screen/sprite. ColourTrans tracks this, and will not recalculate a valid table. Therefore if an application is in any doubt whether the current table is correct, it should request it again; the overheads will be the minimum possible.

3 This returns a byte, representing a colour. This behaviour has been chosen to provide a safe route for those applications which assume that the size of the table in bytes will always be the same as the number of colours in the source mode. In 16bpp, two bytes per colour are returned. In 32bpp a word per colour is returned.

From RISC OS 3.5 onwards a new flag, bit 4 of R5, instructs the call to return >8 bits per colour in a pixel translation table if the destination mode is >8bpp, rather than to return bytes (indicating that the caller is aware that the colours/bytes relationship no longer holds true).

R5 = flags:
bit 4 set ⇒ return > 8 bits per colour rather than bytes

If bit 4 is not set, a table will be returned as if the target mode is 8bpp.

4 This does not generate a look up table. When plotting between these bpp modes only bit stretching/packing is performed.

ColourTrans_SelectTable (page 3-344)
ColourTrans_SelectGCOLTable (page 3-346)
ColourTrans_GenerateTable (page 3-405)

From RISC OS 3.5 onwards, if R2 specifies a mode and R1 is –1, the table uses the default palette for the given mode. This is because the current palette may be unsuitable for the given mode. You can usually get back the old behaviour by using bit 1 of R5.

ColourTrans_SelectGCOLTable (page 3-346)
ColourTrans_GCOLToColourNumber (page 3-366)
ColourTrans_ColourNumberToGCOL (page 3-367)

These calls have not been extended to use 16 or 32 bit GCOL numbers.

ColourTrans_ReadPalette (page 3-393)
ColourTrans_WritePalette (page 3-395)

These calls process palette entries as words which contain 24 bit colour descriptions. The whole palette must be read, modified and written back.

The bottom byte of the palette entry contains the supremacy bits; all 8 bits are reserved. In 32bpp modes bits 7 - 4 are used; in other modes only bit 7 is used. ColourTrans and the kernel now support this. (The kernel only expects one bit of supremacy and ignores the rest.)

The palette entry passed through these calls is in the form &BBGGRRS0, where S is the supremacy mask nibble.

For ColourTrans_ReadPalette, you may set R1 to 0 on entry to make the call use the default palette.

ColourTrans_GenerateTable (page 3-344)

See page 5a-120 and page 5a-121.

New * Command

A new * Command is provided by the new ScreenModes module:

- *LoadModeFile (page 5a-142) loads a ModeInfo file into memory.

For more information see *ModeInfo files* on page 5a-106 and *The ScreenModes module* on page 5a-108.

Changes to existing * Commands

*ScreenLoad (page 1-846)
*ScreenSave (page 1-847)

These calls are now far more likely to cause a mode change, and so reset the graphics window and other state. You should only use these calls to load and save an entire screen, rather than a part of the screen defined by the graphics window.

*WimpMode (page 3-283)

This command now allows the mode to be specified either as a number or as a mode string (see *Mode strings* on page 5a-105). This is reflected in the Display manager application, which also allows this form.

*WimpMode is no longer supported when issued from a task window.

Software vectors

PaletteV (Vector &23)

Called whenever the palette is to be read or written.

The reason codes below have been added in RISC OS 3.5. For information on other reason codes see page 1-105.

On entry

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

Read palette entries

R0 = pointer to word aligned list of logical colours (words), or 0
R1 = type and number of colours:
    bits 0 - 23 = number of palette entries to read
    bits 24 - 31 = type of colour (16,17,18,24 or 25)
R2 = pointer to word aligned buffer to receive 1st flash colour (&BBGGRRxx) – device colours
R3 = pointer to word aligned buffer to receive 2nd flash colour (&BBGGRRxx) – device colours
R4 = 7 (reason code)

Write palette entries

R0 = pointer to word aligned list of logical colours (words), or 0
R1 = type and number of colours:
    bits 0 - 23 = number of palette entries to write
    bits 24 - 31 = type of colour (16,17,18,24 or 25)
R2 = pointer to word aligned list of device colours (&BBGGRRxx)
R4 = 8 (reason code)

Gamma correction tables

R0 = pointer to word aligned gamma correction table for red
R1 = pointer to word aligned gamma correction table for green
R2 = pointer to word aligned gamma correction table for blue
R4 = 9 (reason code)

On exit

Gamma correction tables

R4 = 0 ⇒ the video drivers support gamma correction, and the tables have been copied into system workspace
R4 ≠ 0 ⇒ the video drivers do not support gamma correction

Other reason codes

R4 = 0 ⇒ operation complete

Use

Reason code 7

The memory pointed at by R2 and R3 is filled with words giving the device colour for each flash state. Where only one specific flash state was requested, the information for the other flash state is not filled in.

If no list of logical colours is given (R0 is 0 on entry) and the colour type is 16, 17 or 18, then the call returns the number of palette entries requested starting from the first logical colour – this allows a number of consecutive colours to be read without needing to set up a list.

If the colour type is 16 (read both flash states) and R3 is 0, the area pointed at by R2 is used for both flash states (in the order first state, second state, first state, etc).

Reason code 8

If no list of logical colours is given (R0 is 0 on entry) and the colour type is 16, 17 or 18 on entry then the number of palette entries specified by R1 is written consecutively starting from the first logical colour.

When the colour type is 16 the device colour entries pointed at by R2 should be in the order first state, second state, first state etc.

Reason code 9

This call sets up tables to perform *gamma correction* on RGB values being programmed into the palette. There are three 256-byte tables, one for each of red, green and blue. Before being output to VIDC, the red component of the physical colour (in the range 0 to 255) is used as an index into the red gamma correction table - the value obtained is the gamma corrected red value to be programmed into VIDC. Likewise, the green and blue components are looked up in their respective tables before being output.

Service calls

Service_ModeExtension (Service Call &50)

Allow soft modes

On entry

R1 = &50 (reason code)
R2 = mode specifier that information is requested for
R3 = monitor type (or –1 for don't care)
R4 = memory bandwidth available (in bytes/second)
R5 = total amount of video RAM in system (in bytes)

On exit

All registers preserved (if not claimed)

If claimed:
R1 = 0
R2 preserved
R3 = pointer to VIDC list (type 3)
R4 = pointer to workspace list if mode specifier was a mode number,
or 0 if mode specifier was a pointer to a mode selector

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. **The description below is for RISC OS 3.5 and later only**; for details of this service call under earlier versions of RISC OS, see page 1-641.

In RISC OS it is possible to load modules which provide additional screen modes and additional monitor types. Such modules must claim this call and return the requested information if they recognise the passed mode and monitor type, and if the mode being selected would use no more than the specified video bandwidth and video memory. Otherwise they should pass on the call.

A module that is checking if it recognises a mode selector must examine its format specifier, held in bits 0 - 7 of the flags word (at offset 0). If the module does not recognise the format, it must pass on the service call.

The mode selector could contain –1 as the frame rate, in which case the matching mode with the highest frame rate should be returned.

If R3 holds -1 then RISC OS 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

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. RISC OS 3 supports VIDC lists in formats 0 and 1; these include values that directly correspond to VIDC1 register formats. These formats are **not** supported on RISC OS 3.5 and later; mode extend modules for RISC OS 2 and 3 **will not work**.

VIDC list: format 3

A new format list (type 3) is used from RISC OS 3.5 onwards, which is independent of the video controller used**:**

Offset	Value	
0	3 (format of list)	
4	pixel depth:	0 ⇒ 1bpp, 1⇒ 2bpp, 2⇒ 4bpp 3 ⇒ 8bpp, 4 ⇒ 16bpp, 5⇒ 32bpp
8	horizontal:	sync width (in pixels)
12		back porch (in pixels)
16		left border (in pixels)
20		display size (in pixels)
24		right border (in pixels)
28		front porch (in pixels)
32	vertical:	sync width (in rasters)
36		back porch (in rasters)
40		top border (in rasters)
44		display size (in rasters)
48		bottom border (in rasters)
52		front porch (in rasters)
56	pixel rate (in kHz)	
60	sync polarities when composite sync not configured:	
		bit 0 set ⇒ Hsync inverted bit 1 set ⇒ Vsync inverted bits 2 to 31 reserved (must be zero)
64	video control parameters list	
n	–1 (terminator)	

The video control parameters list (at offset 64) does not normally contain entries for normal video operation. These are only needed for special video operation. The list contains pairs of words (control parameter index, value) terminated by a –1 word. These control additional VIDC registers, bits in registers, and monitor power savings. These are described in the following table. Refer to the VIDC20 data sheet for detailed explanations.

Control index	Parameter description	Values
1	LCD mode	0 ⇒ disable, 1 ⇒ enable
2	LCD dual-panel mode	0 ⇒ disable, 1 ⇒ enable
3	LCD offset register 0	0 - 255
4	LCD offset register 1	0 - 255
5	Hi-res mode	0 ⇒ disable, 1 ⇒ enable
6	DAC control	0 ⇒ disable, 1 ⇒ enable (default)
7	RGB pedestal enables	bit 0 = R, bit 1 = G, bit 2 = B
8	External register [7:0]	0 - 255
9	Reserved	—
10	Reserved	—
11	DPMS power saving	0 - 3; see *Monitor power saving* on page 5a-653

Workspace list

Returning a workspace list is relevant only if a mode number is passed in. If a pointer to a mode selector is passed in, RISC OS works out what the mode variables should be, there is no need to return a workspace list, and R4 is set to zero on exit.

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 the 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 page 5a-123, and *Service_ModeChanging* on page 1-648.

The new computers fitted with VIDC20 vary in their video capabilities. The monitor type, video bandwidth and video RAM parameters allow a mode provider to supply screen modes with identical resolutions but different frame rates, tuned to the particular monitor and computer combination being used. However, any workspace parameters returned **must** be the same, as the mode number is used as an identifier in sprites and in calls such as OS_ReadModeVariable.

This service call is not issued for combinations that RISC OS itself already supports.

Monitor types are allocated by Acorn. There are no monitor types pre-reserved for general use by users.

Service_EnumerateScreenModes (Service Call &8D)

Enumerates the available screen modes

On entry

R1 = &8D (reason code)
R2 = number of modes to skip
R3 = monitor type
R4 = memory bandwidth available (in bytes/sec)
R5 = total amount of video RAM in system (in bytes)
R6 = pointer to block to return data, or 0 to just count entries
R7 = size of block (in bytes) if R6 ≠ 0, or 0 if R6 = 0

On exit

R1 = 0 if claimed (further valid modes are available, but would not fit in block);
else preserved
R2 = – (number of modes filled in)
R3 - R5 preserved
R6 = pointer to byte after last one filled in, or preserved if 0 on entry
R7 = amount of unused space in block,
or – (amount of space needed in block) if R6 = 0 on entry

Use

This service call enumerates the available screen modes. Modules return information on all modes they provide that work on the specified monitor type and which require no more than the specified memory bandwidth and video memory.

OS_ScreenMode 2 provides a front-end for applications (see page 5a-135); you should use it rather than issuing this service call yourself.

By setting R6 and R7 to zero, clients can find the amount of space required to hold all returned modes; they can then issue the call again to actually read the information. Alternatively, clients can use a fixed size buffer, and repeatedly issue the call until it is no longer claimed. When using this method, R2 on entry – the number of modes to skip this iteration – should be set to:

(previous R2 on entry) – (R2 on exit)

This is the same as:

(number of modes skipped last time) + (number of modes filled in this time)

Each mode returned in the block is of the following format:-

Offset	Value
0	size of entry in bytes (24 for this format)
4	mode provider flags: bit 0 = 1 bits 1 - 7 = mode info format specifier (zero for this format) bits 8 - 31 = additional mode info flags (must be zero)
8	x-resolution (in pixels)
12	y-resolution (in pixels)
16	pixel depth (as for mode selector)
20	frame rate (in Hz, to the nearest integer)
24	mode name, null terminated, and then padded with nulls until it is word aligned. (For unnamed modes this will simply be a single word whose value is 0.)

Future modules may use different mode info formats, therefore callers should check bits 0 - 7 of the mode provider flags before extracting the other information in this block. If the caller doesn't recognise the mode info format for an entry, then it can skip the entry by using the size field at offset 0. For format checking purposes, bits 8 - 31 should be ignored.

Mode-providing modules that wish to respond to this service call should use this algorithm:

```
For each mode that they want to return
        If R2 > 0 Then
                do nothing, ie skip it
        Else
                If R6<>0 Then
                        (enumeration case - filling in block)
                        If R7 >= entrysize Then
                                store entry at R6
                                R6 += entrysize
                        Else
                                (not enough space for next mode)
                                R1 = 0 (Service_Serviced)
                                Return (service call claimed)
                        EndIf
                EndIf
                R7 -= entrysize
        EndIf
        R2 -= 1
Next
Return (service call passed on)
```

This service call is only issued under RISC OS 3.5 and later.

SWI calls

OS_SpriteOp 17 (SWI &65)

Checks the validity of a sprite area

On entry

R0 = 17
R1 = pointer to control block of sprite area

On exit

R0, R1 preserved

Use

This checks the validity of a sprite area. Other OS_SpriteOp calls do not make such checks, since it would slow them down too much. Instead it is your application's responsibility to make this call. You would typically call it once after loading a sprite file, to satisfy yourself of the data's integrity. For efficiency, you should not make this call within a redraw loop.

The validation treats offsets as unsigned numbers, and is as follows:

The offset to the first sprite is word aligned, and lies within the 'used' part of the sprite area
The offset to the free area is word aligned, and lies within the sprite area
FOR each sprite
DO The offset to the next sprite is word aligned, and lies within the 'used' part of the sprite area
The first bit used is 0 for a new type sprite, or is in the range 0 - 31 for an old type sprite
The last bit used is in the range 0 - 31
The offset to the image is word aligned, and lies within the sprite
The offset to the mask is word aligned, and lies within the sprite
The space allowed for the sprite image is sufficient to hold an image of the given width, height, and bpp (assumed to be 1bpp if the sprite's mode number is unknown)
The space allowed for the sprite mask is sufficient to hold one of the

given width, height, and bpp (1bpp for a new type sprite, or assumed to be 1bpp if the sprite's mode number is unknown)

OD

If the sprite area is invalid in some way, an error is generated in the usual way for a SWI; the V flag is set on exit, and R0 points to an error block.

Sprites with an unknown mode number are still allowed, because such sprites can usefully occur in sprite files.

These checks do not exclude sprites that conform to the definition of sprite areas, but include unusual features such as an extension area, or an unconventional palette size.

This call is only available from RISC OS 3.6 onwards.

Related SWIs

OS_SpriteOp 10 (page 1-795), OS_SpriteOp 11 (page 1-796)

Related vectors

SpriteV

OS_ScreenMode (SWI &65)

Performs miscellaneous operations for screen mode handling

On entry

R0 = reason code
Other registers depend upon the reason code

On exit

R0 preserved
Other registers depend upon 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 performs miscellaneous operations for screen mode handling.

The particular action of OS_ScreenMode is given by the reason code in R0 as follows:

R0	Action	page
0	Selects a screen mode	5a-135
1	Returns the mode specifier for the current mode	5a-138
2	Enumerates the available screen modes	5a-139
3	Reserved for system use	5a-140

This call is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

OS_ScreenMode 0 (SWI &65)

Selects a screen mode

On entry

R0 = 0 (reason code)
R1 = mode specifier

On exit

All registers preserved

Use

This call selects the given screen mode.

Mode number used

If a mode number *n* is given, then the existing mechanisms are used to select this mode, exactly as if VDU 22,*n* were issued:

- If the mode number is recognised by RISC OS then the mode variables for that mode are loaded from its internal tables. If it is not recognised then Service_ModeExtension is issued; the module which responds to this passes back a workspace list, which contains a base mode (that must be known to RISC OS) and a list of changes to mode variables.
- In certain circumstances a substitute mode can be used

Mode selector used

If a pointer to a mode selector is given, then a new mechanism is used:

- The mode variables are set from the values given in the mode selector block. Any mode variables which are not specified are given sensible defaults, based on the specified x and y resolutions and the pixel depth (see below). Note that RISC OS copies away the relevant information, so mode selector structures need not remain valid after the call has returned.
- If the specified mode cannot be selected for any reason, then an error is always returned.

The default values for any unspecified mode variables are as follows:

Variable	Default value
ModeFlags	0
ScrRCol	(xres >> 3) -1
ScrBRow	(yres >> 3) -1
NColour	1, 3, 15, 63, &FFFF, &FFFFFFFF for pixdepth = 0 to 5 respectively
XEigFactor	1
YEigFactor	1 if yres ≥ xres/2, or 2 if yres < xres/2
LineLength	(xres << pixdepth) >> 3
ScreenSize	((xres × yres) << pixdepth) >> 3
YShftFactor	0
Log2BPP	pixdepth
Log2BPC	pixdepth
XWindLimit	xres-1
YWindLimit	yres-1

Service_ModeExtension still gets issued, but only if RISC OS does not know the video timings for the resolutions/pixel depth/frame rate asked for. The module responding provides only timing and other hardware control information, and not any mode variable values.

In the case where pixdepth=3, the default value of NColour is 63. This means that by default, the palette in 256-colour modes behaves as it does on VIDC1-based machines, i.e. palette entries get modified in groups of 16. This is so that programs which expect the old behaviour work in these modes without modification.

To gain access to fully-palette-programmable 256 colour modes, you should explicitly set these variables:

Variable	Value
ModeFlags	128
NColour	255

All 256 palette entries then become programmable, although they are initially identical to those on a VIDC1-based machine.

You might notice that there is no explicit way of selecting a shadow screen mode. In order to get this effect the program should ensure there is sufficient memory in the screen dynamic area and then switch screen banks.

OS_ScreenMode 1 (SWI &65)

Returns the mode specifier for the current mode

On entry

R0 = 1 (reason code)

On exit

R1 = mode specifier

Use

This call returns the mode specifier for the current screen mode.

If the current screen mode was selected by a mode number then that mode number is returned; otherwise a pointer to a mode selector is returned.

OS_ScreenMode 2 (SWI &65)

Enumerates the available screen modes

On entry

R0 = 2 (reason code)
R2 = value of R2 to pass to Service_EnumerateScreenModes
R6 = value of R6 to pass to Service_EnumerateScreenModes
R7 = value of R7 to pass to Service_EnumerateScreenModes

On exit

R1 = value of R1 returned by Service_EnumerateScreenModes
R2 = value of R2 returned by Service_EnumerateScreenModes
R6 = value of R6 returned by Service_EnumerateScreenModes
R7 = value of R7 returned by Service_EnumerateScreenModes

Use

This call provides a front-end to Service_EnumerateScreenModes (see page 5a-129). It fills in R3 (the current monitor type), R4 (the memory bandwidth available) and R5 (the total amount of video RAM), and then issues the service call.

OS_ScreenMode 3 (SWI &65)

This reason code is for system use only; you must not use it in your own code.

ScreenModes_ReadInfo (SWI &487C0)

Reads the current monitor title

On entry

R0 = 0 (reason code ⇒ read current monitor title)

On exit

R0 = pointer to current monitor title

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call reads the current monitor title, as loaded from the current ModeInfo file. It is used by the Display Manager to show the monitor title in its title bar.

Future versions of RISC OS may add other reason codes to this call.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

* Commands

*LoadModeFile

Loads a ModeInfo file into memory

Syntax

`*LoadModeFile` *`filename`*

Parameters

`filename`	a valid pathname specifying a file

Use

This command loads a ModeInfo file into memory. If the file contains valid information, it sets the current monitor type to 7 (file). This then makes available all the screen modes defined in the file, while removing all modes defined in any previously loaded file.

This command is only available from RISC OS 3.5 onwards.

Example

```
*LoadModeFile adfs::MHardy.$.Modes.AKF50
```

Related commands

None

*VIDCBandwidthLimit

This command is for internal use only; you must not use it in your own code.

This command is only available from RISC OS 3.5 onwards.

**VIDCBandwidthLimit*

107 JPEG images

Introduction and Overview

The SpriteExtend module has been extended in RISC OS 3.6 to support JPEG images through a SWI interface.

JPEG is an international standard data format for the lossy compression of photographic data, capable of encoding colour images at screen resolutions using about 1½ - 2½ bits per pixel.

Because of the compression used, many of the operations you can perform on uncompressed bitmaps – such as sprites – are difficult or impossible to perform on JPEG images. This includes operations such as adding or deleting rows or columns, and arbitrary transformations. The support provided for JPEG images is therefore restricted in RISC OS 3.6 to providing information on them, and simple scaled plotting and printing.

The CompressJPEG module

A separate CompressJPEG module provides SWIs with which you can compress raw data into a JPEG image. See the chapter *CompressJPEG* on page 5a-617 for details both of the compression and decompression algorithms used with JPEGs, and of the SWIs it provides.

Technical details

SWI naming and numbering

Although the SpriteExtend module provides the JPEG SWIs, they use their own SWI chunk (base number &49980) and SWI name prefix ('JPEG_'). The SWIs are described below, from page 5a-149 onwards.

The JPEG standard

The JPEG standard is split into two parts:

- ISO DIS 10918-1, Digital Compression and Coding of Continuous-tone Still Images, Part 1: Requirements and guidelines.
- ISO DIS 10918-2, Digital Compression and Coding of Continuous-tone Still Images, Part 2: Compliance testing.

At the time of going to press, Part 2 was still in draft stage.

You may also find these references useful:

- The JPEG Still Picture Compression Standard / Gregory K Wallace
  in
  IEEE Transactions on Consumer Electronics, December 1991.
- JPEG Still Image Data Compression Standard / William B. Pennebaker; Joan L. Mitchell. – New York, USA: Van Nostrand Reinhold, 1993.

JFIF files

The JPEG standard is wide-ranging in its scope, and allows many bizarre parameters and combinations. To limit these to more reasonable proportions various subsets of the standard have been defined. By far the most popular of these is the JFIF (JPEG File Interchange Format) standard, defined by C-Cube. This is widely used for the simple interchange of JPEG data; indeed, when people talk about 'JPEG files', they usually mean JFIF files.

The code in RISC OS modules only supports images that conform to version 1.02 or earlier of the JFIF standard. JFIF files are allocated the file type &C85; the textual equivalent is 'JPEG'. The sprite for this file type is included in the Wimp sprite pool.

Documentation of the JFIF standard is available as follows:

- JPEG File Interchange Format (JFIF) / Eric Hamilton – version 1.02 – C-Cube Microsystems, 1778 McCarthy Blvd, Miltipas, CA 95035.

ChangeFSI and JPEG files

ChangeFSI can output JPEG files; these all conform to the JFIF standard.

ChangeFSI also accepts JPEG files as input. If a file only uses the JFIF subset of the JPEG standard, ChangeFSI fully understands it, and so correctly processes it. If a file uses features that are excluded from the JFIF subset, about which ChangeFSI does not know, it will make assumptions. Sometimes these will be correct, and so the file will be correctly processed; otherwise the file will be incorrectly processed.

Hence **you cannot use ChangeFSI to test for JFIF-conformance**. Some images that ChangeFSI correctly processes may be faulted by the SpriteExtend JPEG SWIs as not conforming to the JFIF standard.

Dithering of JPEGs

When you call the JPEG plotting SWIs you can set bit flags to request that when plotting to a shallow screen mode the output is dithered, with or without error diffusion. Three types of dithering are used:

Ordered dither

This is the simplest form of dithering available; it displays colours that are unavailable by using small patterns made up of the closest available colours. This is the default form of dithering, used in most cases when the dithering bit is set.

YUV error diffused dither

However, when decompressing a JPEG image into an 8bpp mode with the standard palette, an optimised mode is used. This uses a limited error diffusion technique directly on the YUV data in the JPEG, which vastly improves the appearance of the image. This technique will only work on JPEG images which have been compressed using an X and Y sample size of 2, as created both by the official Independent Group's software and by versions 1.03 onwards of ChangeFSI.

It is thus possible that two apparently similar JPEG images can give quite different display qualities because they are compressed differently, and so RISC OS can only apply YUV error diffusion to one of them.

Full error diffused dither

If you set both the dithering and error diffusion bits, then this slower but more accurate form of dithering is used. Speed and space considerations mean that the output image will still not be quite so high a quality as ChangeFSI can produce.

Under RISC OS 3.6 full error diffused dithering can only be used when plotting to an 8bpp screen mode.

SWI Calls

JPEG_Info (SWI &49980)

Gives information on a JPEG image held in a buffer

On entry

R0 = flags for desired operation:
    bit 0 set ⇒ return dimensions, clear ⇒ don't return dimensions
    all other bits reserved (must be zero)
R1 = pointer to buffer holding JPEG image
R2 = length of JPEG image, in bytes

On exit

R0 = returned information flags:
    bit 0 set ⇒ greyscale image, clear ⇒ colour image
    bit 1 set ⇒ transformed plots not supported, clear ⇒ supported
    bit 2 set ⇒ pixel density is a simple ratio, clear ⇒ pixel density is in dpi
R1 preserved
R2 = width, in pixels (if R0 bit 0 set on entry)
R3 = height, in pixels (if R0 bit 0 set on entry)
R4 = x pixel density
R5 = y pixel density
R6 = SpriteExtend's additional extra workspace requirements to plot JPEG
    (0 ⇒ no additional extra workspace required)

Interrupts

Interrupt status is undefined
FIQs are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call gives information on a JPEG image held in a buffer in memory.

It checks the header enough to return the width and height, and does a partial validation of the data. It returns an error if the image appears to be invalid; if no error is returned you may assume that the data is a JPEG image.

This call is only available from RISC OS 3.6 onwards.

Related SWIs

JPEG_FileInfo (page 5a-151)

Related vectors

None

JPEG_FileInfo (SWI &49981)

Gives information on a JPEG image held in a file

On entry

R0 = flags for desired operation:
bit 0 set ⇒ return dimensions, clear ⇒ don't return dimensions
all other bits reserved (must be zero)
R1 = pointer to pathname of JPEG file, control character terminated

On exit

R0 = returned information flags:
bit 0 set ⇒ greyscale image, clear ⇒ colour image
bit 1 set ⇒ transformed plots not supported, clear ⇒ supported
bit 2 set ⇒ pixel density is a simple ratio, clear ⇒ pixel density is in dpi
R1 preserved
R2 = width, in pixels
R3 = height, in pixels
R4 = x pixel density
R5 = y pixel density
R6 = SpriteExtend's additional extra workspace requirements to plot JPEG
(0 ⇒ no additional extra workspace required)

Interrupts

Interrupt status is undefined
FIQs are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call gives information on a JPEG image held in a file.

This call checks the header enough to return the width and height, and does a partial validation of the data. It returns an error if the image appears to be invalid; if no error is returned you may assume that the data is a JPEG image.

This call is only available from RISC OS 3.6 onwards.

Related SWIs

JPEG_Info (page 5a-149)

Related vectors

None

JPEG_PlotScaled (SWI &49982)

Decompresses, scales, and plots on the screen a JPEG image held in a buffer

On entry

R0 = pointer to buffer holding JPEG image
R1 = x coordinate at which to plot
R2 = y coordinate at which to plot
R3 = pointer to scale factors (see page 1-780): 0 ⇒ no scaling
R4 = length of JPEG image, in bytes
R5 = flags:
bit 0 set ⇒ dither output when plotting 24 bit JPEG at 16bpp or below
bit 1 set ⇒ dithering (if any) is full error diffused when plotting at 8bpp
all other bits reserved (must be zero)

On exit

R0 - R5 preserved

Interrupts

Interrupt status is undefined
FIQs are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI decompresses, scales, and plots on the screen a JPEG image held in a buffer in memory.

The functionality of this call for JPEGs is similar to that of OS_SpriteOp 52 (Put sprite scaled - see page 1-830) for sprites. The scale factors and the coordinates have exactly the same meaning, and the scaling algorithms used are the same in both calls. However, this call only supports a direct plot (ie plot action 0 of OS_SpriteOp 52).

In plotting the JPEG, the SpriteExtend module may claim extra workspace in a dynamic area to store tables etc. It keeps these cached until either it is asked to plot a different JPEG, or the user decreases the dynamic area's size. This speeds up successive replots of the same JPEG. You can find how much extra workspace SpriteExtend will require – if any – by first calling JPEG_Info (page 5a-149) or JPEG_FileInfo (page 5a-151). You can hence ensure there is sufficient free memory before making this call.

This call returns an error if it cannot claim sufficient memory to plot the JPEG image, or if the image appears incomplete or corrupt in some way.

This call is only available from RISC OS 3.6 onwards.

Related SWIs

JPEG_Info (page 5a-149), JPEG_FileInfo (page 5a-151)

Related vectors

None

JPEG_PlotFileScaled (SWI &49983)

Decompresses, scales, and plots on the screen a JPEG image held in a file

On entry

R0 = pointer to pathname of JPEG file, control character terminated
R1 = x coordinate at which to plot
R2 = y coordinate at which to plot
R3 = pointer to scale factors (see page 1-780): 0 ⇒ no scaling
R4 = flags:
bit 0 set ⇒ dither output when plotting 24 bit JPEG at 16bpp or below
bit 1 set ⇒ dithering (if any) is full error diffused when plotting at 8bpp
all other bits reserved (must be zero)

On exit

R0 - R4 preserved

Interrupts

Interrupt status is undefined
FIQs are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI decompresses, scales, and plots on the screen a JPEG image held in a file.

The functionality of this call for JPEGs is similar to that of OS_SpriteOp 52 (Put sprite scaled – see page 1-830) for sprites. The scale factors and the coordinates have exactly the same meaning, and the scaling algorithms used are the same in both calls. However, this call only supports a direct plot (ie plot action 0 of OS_SpriteOp 52).

The file is loaded into memory for the duration of the call, but is not cached. This call therefore uses as much memory as loading the file into a buffer yourself and then calling JPEG_PlotScaled (page 5a-153), and gives you no control over whether the image remains cached. Furthermore, although the mechanism exists to pass this call on to the printer drivers, they do not support it. We therefore strongly recommend that you use JPEG_PlotScaled in preference to this call.

As well as the memory to hold the file, the SpriteExtend module may claim extra workspace in a dynamic area to store tables etc. It keeps these cached until either it is asked to plot a different JPEG, or the user decreases the dynamic area's size. This speeds up successive replots of the same JPEG. You can find how much extra workspace SpriteExtend will require – if any – by first calling JPEG_Info (page 5a-149) or JPEG_FileInfo (page 5a-151). You can hence ensure there is sufficient free memory before making this call: enough both to hold the file, and to provide any extra workspace required.

This call returns an error if it cannot claim sufficient memory to plot the JPEG image, or if the image appears incomplete or corrupt in some way.

This call is only available from RISC OS 3.6 onwards.

Related SWIs

JPEG_PlotScaled (page 5a-153)

Related vectors

None

JPEG_PlotTransformed (SWI &49984)

Decompresses, transforms, and plots on the screen a JPEG image held in a buffer

On entry

R0 = pointer to buffer holding JPEG image
R1 = flags:
bit 0 set ⇒ R2 = pointer to destination coordinate block, else to matrix
bit 1 set ⇒ dither output when plotting 24 bit JPEG at 16bpp or below
bit 2 set ⇒ dithering (if any) is full error diffused when plotting at 8bpp
all other bits reserved (must be zero)
R2 = pointer to destination coordinate block (if R2 bit 0 set), or
pointer to Draw-style transformation matrix (if R2 bit 0 clear)
R3 = length of JPEG image, in bytes

On exit

R0 - R3 preserved

Interrupts

Interrupt status is undefined
FIQs are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI decompresses, transforms, and plots on the screen a JPEG image held in a buffer in memory.

The functionality of this call for JPEGs is similar to that of OS_SpriteOp 56 (Put sprite transformed – see page 1-833) for sprites. The destination coordinate block and the transformation matrix have exactly the same meaning. However, this call only supports a direct plot (ie plot action 0 of OS_SpriteOp 56).

Under RISC OS 3.6 this call only supports simple scaling, with no rotation or other transformation involved. Any unsupported transformation gives an appropriate error.

In plotting the JPEG, the SpriteExtend module may claim extra workspace in a dynamic area to store tables etc. It keeps these cached until either it is asked to plot a different JPEG, or the user decreases the dynamic area's size. This speeds up successive replots of the same JPEG. You can find how much extra workspace SpriteExtend will require – if any – by first calling JPEG_Info (page 5a-149) or JPEG_FileInfo (page 5a-151). You can hence ensure there is sufficient free memory before making this call.

This call returns an error if it cannot claim sufficient memory to plot the JPEG image, or if the image appears incomplete or corrupt in some way.

This call is only available from RISC OS 3.6 onwards.

Related SWIs

JPEG_Info (page 5a-149), JPEG_FileInfo (page 5a-151), JPEG_PlotScaled (page 5a-153)

Related vectors

None

JPEG_PlotFileTransformed (SWI &49985)

Decompresses, transforms, and plots on the screen a JPEG image held in a file

On entry

R0 = pointer to pathname of JPEG file, control character terminated
R1 = flags:
bit 0 set ⇒ R2 = pointer to destination coordinate block, else to matrix
bit 1 set ⇒ dither output when plotting 24 bit JPEG at 16bpp or below
bit 2 set ⇒ dithering (if any) is full error diffused when plotting at 8bpp
all other bits reserved (must be zero)
R2 = pointer to destination coordinate block (if R2 bit 0 set), or
pointer to Draw-style transformation matrix (if R2 bit 0 clear)

On exit

R0 - R2 preserved

Interrupts

Interrupt status is undefined
FIQs are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI decompresses, transforms, and plots on the screen a JPEG image held in a file.

The functionality of this call for JPEGs is similar to that of OS_SpriteOp 56 (Put sprite transformed – see page 1-833) for sprites. The destination coordinate block and the transformation matrix have exactly the same meaning. However, this call only supports a direct plot (ie plot action 0 of OS_SpriteOp 56).

Under RISC OS 3.6 this call only supports simple scaling, with no rotation or other transformation involved. Any unsupported transformation gives an appropriate error.

The file is loaded into memory for the duration of the call, but is not cached. This call therefore uses as much memory as loading the file into a buffer yourself and then calling JPEG_PlotTransformed (page 5a-157), and gives you no control over whether the image remains cached. Furthermore, although the mechanism exists to pass this call on to the printer drivers, they do not support it. We therefore strongly recommend that you use JPEG_PlotTransformed in preference to this call.

As well as the memory to hold the file, the SpriteExtend module may claim extra workspace in a dynamic area to store tables etc. It keeps these cached until either it is asked to plot a different JPEG, or the user decreases the dynamic area's size. This speeds up successive replots of the same JPEG. You can find how much extra workspace SpriteExtend will require – if any – by first calling JPEG_Info (page 5a-149) or JPEG_FileInfo (page 5a-151). You can hence ensure there is sufficient free memory before making this call: enough both to hold the file, and to provide any extra workspace required.

This call returns an error if it cannot claim sufficient memory to plot the JPEG image, or if the image appears incomplete or corrupt in some way.

This call is only available from RISC OS 3.6 onwards.

Related SWIs

JPEG_PlotTransformed (page 5a-157)

Related vectors

None

JPEG_PDriverIntercept (SWI &49986)

Requests that SpriteExtend passes on all calls to JPEG plotting SWIs

On entry

R0 = flags:
bit 0 set ⇒ pass on plotting calls, clear ⇒ don't pass on plotting calls
bit 1 set ⇒ use translation tables, clear ⇒ don't use translation tables
all other bits reserved (must be zero)

On exit

R0 = previous intercept state

Interrupts

Interrupt status is undefined
FIQs are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI is used by the printer drivers to request that SpriteExtend passes on all calls to JPEG plotting SWIs by itself calling PDriver_JPEGSWI (see page 5a-587). When SpriteExtend passes on these calls, it ignores them itself.

You must not make this call from your own applications.

The JPEG plotting SWIs (ie those that are passed on) are listed in *Related SWIs* below.

Related SWIs

JPEG_PlotScaled (page 5a-153), JPEG_PlotFileScaled (page 5a-155)
JPEG_PlotTransformed (page 5a-157), JPEG_PlotFileTransformed (page 5a-159)

Related vectors

None

108 Miscellaneous kernel items

Introduction and Overview

This chapter describes some minor changes that do not belong in any of the previous chapters about the kernel.

Changes to existing SWIs

OS_Byte 129 (page 1-899)

When reading the OS version identifier, R1 returns on exit the value:

- &A5 for RISC OS 3.5
- &A6 for RISC OS 3.6.

New SWI

A SWI has been added in RISC OS 3.5 to reset the computer. It is described overleaf.

SWI Calls

OS_Reset (SWI &6A)

Performs a hard reset

On entry

—

On exit

Does not exit!

Interrupts

Interrupt state is not defined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Irrelevant

Use

This call performs a hard reset.

It is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

Part 16 – Filing and networking

109 FileSwitch

Introduction and Overview

Under RISC OS 3.6 FileSwitch has been extended to support larger capacity storage devices, such as those now supported by FileCore-based filing systems.

The only change made has been to provide three new reason codes for OS_FSControl (page 2-80), each of which duplicates previously available functionality, but allows 64 bit values to be passed or returned instead of 32 bit values.

New OS_FSControl reason codes (page 2-80)

The three new reason codes are:

R0	Action	Page
55	Read the free space on the disc or image file that holds a specified object	5a-168
56	Return the defect list for an image	5a-169
57	Map out a defect from an image	5a-170

New filing system entry points

For each of the new OS_FSControl reason codes, a corresponding new reason code has been added to those that may be passed to a filing system's FSEntry_Func entry point and to an image filing system's ImageEntry_Func entry point.

If you are writing a filing system, and hence need to know the details of these new reason codes, you should see *Writing a filing system* on page 5a-261.

SWI Calls

OS_FSControl 55 (SWI &29)

Reads the free space on the disc or image file that holds a specified object

On entry

R0 = 55 (reason code)
R1 = pointer to name of object (null terminated)

On exit

R0 = bits 0 - 31 of free space
R1 = bits 32 - 63 of free space
R2 = largest creatable object
R3 = bits 0 - 31 of disc size
R4 = bits 32 - 63 of disc size

Use

This call reads the free space on the disc or image file that holds the specified object. It also returns the size of the largest creatable object, and the size of the disc.

This call is similar to OS_FSControl 49 (page 2-134), except the values for disc size and free space returned are 64 bit values. If an error occurs, this may mean the filing system does not support this call, in which case you should then try OS_FSControl 49.

This call is only available from RISC OS 3.6 onwards, and returns incorrect information for NetFS.

OS_FSControl 56 (SWI &29)

Returns the defect list for an image

On entry

R0 = 56 (reason code)
R1 = pointer to name of image (null terminated)
R2 = pointer to buffer
R5 = buffer length

On exit

R0 preserved
R1 = number of defects placed in buffer
R2, R5 preserved

Use

This call fills the given buffer with a defect list, which gives the byte offset to the start of each defect. Each entry in the list is a pair of words – with the least significant one first – giving the address of the defect as a 64 bit value.

This call is similar to OS_FSControl 41 (page 2-126). If an error occurs, this may mean the filing system does not support this call, in which case you should then try OS_FSControl 41.

This call is only available from RISC OS 3.6 onwards, and returns incorrect information for NetFS.

OS_FSControl 57 (SWI &29)

Maps out a defect from an image

On entry

R0 = 57 (reason code)
R1 = pointer to name of image (null terminated)
R2 = bits 0 - 31 of offset to start of defect
R3 = bits 32 - 63 of offset to start of defect

On exit

R0 - R2 preserved

Use

This call maps out a defect from the given image.

This call is similar to OS_FSControl 42 (page 2-127), except the offset to the defect is passed as a 64 bit value. If an error occurs, this may mean the filing system does not support this call, in which case you should then try OS_FSControl 42.

This call is only available from RISC OS 3.6 onwards, and returns incorrect information for NetFS.

110 FileCore

Introduction and Overview

Under RISC OS 3.5 and earlier, FileCore-based filing systems can support 4 hard discs, each with a maximum size of 512 MB, giving a maximum storage capacity per filing system of 2 GB.

This limitation means that using hard discs larger than 512 MB would involve mounting separate partitions as ‘discs’; and using discs larger than 2 GB would require multiple filing systems per disc.

With the continuing push by hard disc manufacturers to reduce the cost per megabyte of their devices, drives with less than 512 MB capacity will cease to be the most cost-effective ones. Furthermore, higher capacity drives are now becoming readily available at affordable prices.

Clearly the limitations of FileCore-based filing systems are becoming increasingly restrictive. RISC OS 3.6 introduces extensions to the logical format of FileCore-based filing systems that remove many of these restrictions, as a result of which:

- the recommended maximum hard disc size is 4 GB
- the maximum size of a file (and hence of an image filing system) is 2^{31}–1 bytes (2 GB)
- the maximum number of disc objects remains 2^{15}–2 (32766).

FileSwitch based systems remain as before:

- the maximum hard disc size is dependent on the underlying file system
- the maximum size of a file (and hence of an image filing system) is 2^{32}–1 bytes (4 GB)
- the maximum number of disc objects is dependent on the underlying file system.

Technical details

Disc record

The disc record (page 2-204) has been extended to support large discs. This uses some of the reserved bytes at the end of the record, the tail end of which now is as follows:

Offset	Name	Meaning	
36	*disc_size_2*	Most significant word of disc size, in bytes	
40	*share_size*	bits 0 - 3:	$\log_2$ (sharing granularity in sectors)
		bits 4 - 7:	reserved – must be zero
41	*big_flag*	bit 0:	set ⇒ RISC OS partition is > 512 MB
		bits 1 - 7:	reserved – must be zero
42 - 59		Reserved – must be zero	

The *disc_size_2* field gives the most significant word of the disc size, and so is used for discs of over 4 GB. (The least significant word is held in the *disc_size* field.)

The *big_flag* bit is so FileCore can tell at mount time whether or not the RISC OS filing system on the disc is big (ie > 512 MB in size), and hence whether or not it uses the new logical format. It cannot use the *disc_size* fields for this, since a disc of over 512 MB may only have a small RISC OS partition, and use the rest of the disc for RISC *i*X.

The *share_size* field controls the granularity of sharing. See *Internal disc addresses* on page 5a-173.

Disc addresses

Physical disc addresses

FileCore performs all low-level disc access through two entry points – DiscOp and MiscOp – provided by FileCore modules such as ADFS. Disc addresses are passed to these entry points as a 32 bit quantity.

Under RISC OS 3.5 and earlier, the physical disc address (page 2-210) combines both the drive number (0 - 7, held in bits 29 - 31), and the byte offset into the disc (0 - 512 MB, held in bits 0 - 28). It is this offset field that restricts FileCore discs to a maximum size of 512 MB.

However, some bits in the offset are redundant, since all programmer interfaces use sector aligned addresses. In RISC OS 3.6 FileCore has been enhanced to make use of these bits; it now also supports disc addresses where the offset is given in sectors, rather

than in bytes. So with the resulting 29 bit sector number, and a sector size of 512 bytes (as typically used on IDE hard discs), this gives a maximum theoretical disc size of $2^{29} \times 512$ bytes, or 256 GB.

Internal disc addresses

FileCore can share the use of a disc object on a new map disc (ie a logical group of fragments) between many objects (ie files or directories). The objects must either be a directory and files within that directory, or files that have the same parent directory. There may not be more than one directory in any disc object, since the directory must always be at the start of the disc object.

New map discs use an internal disc address (see page 2-211) to refer to shared objects, specifying them in terms of their fragment id (0 - &7FFF), and their offset within the disc object (0 - 254, stored as 1 - 255). Under RISC OS 3.5 and earlier, the offset is in units of sectors, which with a 512 byte sector size corresponds to 0 - 127 KB. Thus FileCore can only share the first 254 sectors (127 KB for our example) of a shared disc object, and if the smallest fragment size is larger than this FileCore cannot share all the space in shared disc objects.

From RISC OS 3.6 onwards, you can increase the granularity of the offset within the disc object. It now gives the offset in units of 2^{share_size} sectors, where *share_size* comes from the disc record. You should ensure that if you format a disc, *share_size* is sufficiently large for the following to be true:

$$\text{smallest fragment size} \leq (254 \times 2^{log2secsize} \times 2^{share\_size})$$

FileCore can then share all the space within a shared disc object.

Defect list

The defect list (page 2-215) has been extended by appending a second defect list containing all defects more than 512 MB from the start of the disc. The list works similarly to the first one, but all disc addresses are stored as absolute sector numbers, and the final marker word is &400000*yy*. The byte yy is a check-byte calculated from the previous words in the second defect list only. It is calculated in the same way as the check-byte for the first defect list. Thus a new defect list would look like:

```
defect byte address(es)
&200000xx
defect sector address(es)
&400000yy
```

An empty defect list would now be:

```
&20000000
&40000000
```

To determine whether the second defect list is present, you should examine the *big_flag* byte (see *Disc record* on page 5a-172). If bit 0 is set then the second defect list must be present.

Maximum practical disc size

As discs get larger, so does the smallest fragment size required to format them. For example on a 4 GB disc the smallest fragment size rises to 128K, for two reasons:

- The maximum length of a new map is 64 KB, because the FreeLink field in the map block header (see *Header* on page 2-203) must be able to point to the end of the map, and is only two bytes long. There are hence 512K (64K × 8) allocation bits in the map.

  The allocation size is disc size / allocation bits, which is 4GB / 512K, or 8 KB.

  From page 2-206, the smallest fragment size is (*idlen* + 1) × allocation unit, which is (15 + 1) × 8 KB, or 128 KB.
- The number of possible fragment ids is 2^{idlen}. Since idlen cannot exceed 15, this is a maximum of 2^{15}.

  Each fragment must have available its own fragment id. The smallest fragment size is therefore (disc size / maximum fragment id), which is 4 GB / 2^{15}, or 128 KB.

You can use larger discs, but with smallest fragment sizes of 256 KB or more, you are likely to waste high proportions of disc space in normal use. For this reason we don't recommend you do so.

Disc formats

D format hard discs **are not supported** from RISC OS 3.5 onwards.

The RISC OS 3.6 version of FileCore supports all current E and F format discs.

Changes to existing SWIs

FileCore_DiscOp (page 2-223)

With previous versions of FileCore, on exit from the call, R2 contains the disc address of the next byte to be transferred. From RISC OS 3.6 onwards the address returned is rounded down to be sector aligned.

FileCore_Create (page 2-228)

The descriptor block pointed to by R0 has had a new flag bit added to indicate support for sector addressing. See *Descriptor block (page 2-597)* on page 5a-265.

FileCore_FreeSpace (page 2-231)

The values this call returns may now be too large to represent in a single register. To avoid such problems, the values returned are limited to a maximum of &7FFFFFFF, which you may take to mean ‘at least 2 GB’.

See also FileCore_FreeSpace64 (page 5a-183), a new SWI which provides facilities for returning 64 bit values.

New SWIs

The following new SWIs have been added to FileCore:

- FileCore_MiscOp has had two reason codes added:

  Reason code 6 (page 5a-176) reads the information passed in a descriptor block when creating a new instantiation of a FileCore based filing system. The main use of this is to determine whether a filing system supports sector addressing, or only byte addressing.

  Reason code 7 (page 5a-177) returns the status of a drive; under RISC OS 3.6, this information is restricted to whether or not a drive is locked.
- FileCore_SectorOp (page 5a-178) provides the same functionality as FileCore_DiscOp, save that it uses sector addresses rather than byte addresses.
- FileCore_FreeSpace64 (page 5a-183) provides the same functionality as FileCore_FreeSpace, but uses 64 bit values rather than 32 bit ones.

SWI Calls

FileCore_MiscOp 6 (SWI &40549)

Reads information from a FileCore module's descriptor block

Entry

R0 = 6 (reason code)
R8 = pointer to FileCore instance private word.

Exit

R0 = pointer to block:

Offset	Contains
0	bit flags from FileCore module's descriptor block
3	filing system number
4	address of filing system title
8	address of boot text
12	address of low-level disc op entry
18	address of low-level miscellaneous entry

Use

This call reads information from a FileCore module's descriptor block (page 2-597 and page 5a-265), as passed to FileCore_Create (page 2-228). However this call returns addresses, rather than the offsets into the module that you pass to FileCore_Create.

The main use of this call is to determine whether a filing system supports sector addressing (bit 2 of the bit flags is set), or only byte addressing (bit 2 of the bit flags is clear).

This call is only available from RISC OS 3.6 onwards.

FileCore_MiscOp 7 (SWI &40549)

Returns the status of the given drive

Entry

R0 = 7 (reason code)
R1 = drive number
R8 = pointer to FileCore instance private word.

Exit

R2 = flag word:
bit 0 set ⇒ drive is locked
all other bits reserved

Use

This call returns the status of the given drive. It can be called in the background

The main use of this call is so that FileCore can cleanly check whether or not a drive is locked before restarting a background read or write operation.

This call is only available from RISC OS 3.6 onwards.

FileCore_SectorOp (SWI &4054A)

Performs various operations on a disc using sector addressing

On entry

R1 bits 0 - 3 = reason code
bits 4 - 7 = option bits
bits 8 - 31 = bits 2 - 25 of pointer to alternative disc record, or zero
R2 = disc address
R3 = pointer to buffer
R4 = length in bytes
R6 = cache handle
R8 = pointer to FileCore instance private word

On exit

R1 preserved
R2 = disc address of next sector to which to transfer
R3 = pointer to next buffer location to be transferred
R4 = number of bytes not transferred

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call performs various disc operations as specified by bits 0 - 3 of R1:

Value	Meaning		Uses	Updates
0	Verify		R2, R4	R2, R4
1	Read sectors		R2, R3, R4	R2, R3, R4
2	Write sectors		R2, R3, R4	R2, R3, R4
3	Floppy disc: read track		R2, R3	
	Hard disc: read Id		R2, R3	
4	Write track		R2, R3	
5	Seek (used only to park)		R2	
6	Restore		R2	
7	Floppy disc: step in	†		
8	Floppy disc: step out	†		
9	Read sectors via cache		R2, R3, R4, R6	R2, R3, R4, R6
15	Hard disc: specify		R2	

† These reason codes are only valid with the 1772 disc controller. They are not supported on 710/711 based machines (such as the A5000) and should be avoided for future compatibility.

This call provides the same functionality as FileCore_DiscOp (page 2-223), save that it uses sector addresses rather than byte addresses. It is only available from RISC OS 3.6 onwards.

Option bits

The option bits have the following meanings:

Bit 4

This bit is set if an alternate defect list for a hard disc is to be used. This is assumed to be in RAM 64 bytes after the start of the disc record pointed to by bits 8 - 31 of R1 shifted left 6 bits (so they form bits 2 - 25 of the pointer).

This bit may only be set for old map discs.

Bit 5

If this bit is set, then the meaning of R3 is altered. It does not point to the area of RAM to or from which the disc data is to be transferred. Instead, it points to a word-aligned list of memory address/length pairs. All but the last of these lengths must be a multiple of the sector size. These word-pairs are used for the transfer until the total number of bytes given in R4 has been transferred.

On exit, R3 points to the first pair which wasn't fully used, and this pair is updated to reflect the new start address/bytes remaining, so that a subsequent call would continue from where this call has finished.

This bit may only be set for reason codes 0 - 2.

Bit 6

If this bit is set then escape conditions are ignored during the operation, otherwise they cause it to be aborted.

Bit 7

If this bit is set, then the usual timeout for floppy discs of 1 second is not used. Instead FileCore will wait (forever if necessary) for the drive to become ready.

Disc address

The disc address is a sector offset from the start of the disc. It must be on a track boundary for reason codes other than 0-2 and 9. Note that you must make allowances for any defects, as the disc address is not corrected for them.

For reason code 6 (restore), the disc address is only used for the drive number; the bottom 29 bits should be set to zero.

Where the transfer length is not a multiple of the sector size, the end disc address specifies the sector holding the byte after the last one that was transferred.

The *specify disc* command (reason code 15) sets up the defective sector list, hardware information and disc description from the disc record supplied. Note that in memory, this information must be stored in the order disc record, then defect list/hardware parameters.

Read Track/ID (reason code 3)

If the alternate defect list option bit (bit 4) is set in R1 on entry when reading a track/ID, then a whole track's worth of ID fields is read. This usage is not available under RISC OS 2.

The call reads 4 bytes of sector ID information into the buffer pointed to by R3 for every sector on the track. The order of data is:

Cylinder
Head
Sector number
Sector size (0= 128, 1= 256, etc)

For floppy discs, the operation is terminated after 200mS (1 revolution).

The first sector ID transferred will normally be that following the index mark (it may be the second if there is abnormal interrupt latency from the index pulse interrupt). The first two ID's read may also be duplicated at the buffer end due to interrupt latency. Consequently the buffer should be at least 16 bytes longer than the maximum number of IDs expected (512 bytes at most).

The disc record provided is updated to return the actual number of sectors per track found (at offset 1). Note to use this option you **must** provide a valid defect list following on after the disc record. The minimal defect list is a word of &20000000 for small discs (ie byte addressed), or two words of &20000000 followed by &40000000 for large discs (ie sector addressed).

Write Track (reason code 4)

If R3 (the buffer pointer) is non-zero on entry, this reason code is used to write a track. This usage is specific to the 1772 disc controller.

If R3 is zero on entry, this reason code is instead used to format a track; R4 then points to a disc format structure. This usage is available with all controllers, but is not available under RISC OS 2.

The disc format structure pointed to by R4 is as follows:

Offset	Length	Meaning
0	4	Sector size in bytes (which must be a multiple of 128)
4	4	Gap1
8	4	Reserved – must be zero
12	4	Gap3
16	1	Sectors per track
17	1	Density: 1 single density (125Kbps FM) 2 double density (250Kbps FM) 3 double+ density (300Kbps FM) (ie higher rotation speed double density) 4 quad density (500Kbps FM) 8 octal density (1000Kbps FM)
18	1	Options: bit 0 1 index mark required

		bit 1	1	double step
		bits 2-3	0	interleave sides
			1 - 3	sequence sides
		bits 4-7		reserved – must be 0
19	1	Sector fill value		
20	4	Cylinders per drive (normally 80)		
24	12	Reserved – must be 0		
36	?	Sector ID buffer, 1 word per sector:		
		bits 0 - 7		Cylinder number mod 256
		bits 8 - 15		Head (0 for side 1, 1 for side 2)
		bits 16 - 23		Sector number
		bits 24 - 31		Log_2 (sector size) – 7, eg 1 for 256 byte sector

An error is generated if the specified format is not possible to generate, or if the track requested is outside the valid range. The tracks are numbered from 0 to (number of tracks) – 1. The mapping of the address is controlled by the disc structure record.

Read sectors via cache (reason code 9)

This reason code reads sectors via a cache held in the RMA. It is not available under RISC OS 2.

To start a sequence of these operations, set R6 (the cache handle) to zero on entry. Its value will be updated on exit, and subsequent calls should use this new value.

Bits 4 - 7 of R1 should be zero, and are ignored if set.

To discard the cache once finished, call FileCore_DiscardReadSectorsCache (see page 2-235).

Related SWIs

None

Related vectors

None

FileCore_FreeSpace64 (SWI &4054B)

Returns 64 bit information on a disc's free space

On entry

R0 = pointer to disc specifier (null terminated)
R8 = pointer to FileCore instance private word

On exit

R0 = bits 0 - 31 of total free space on disc
R1 = bits 32 - 63 of total free space on disc
R2 = size of largest object that can be created, or &7FFFFFFF if 2 GB or more

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 total free space on the given disc, and the largest object that can be created on it. As with FileCore_FreeSpace (see page 5a-175), the returned *size of largest object* is restricted to a maximum of &7FFFFFFF, meaning 'at least 2 GB'.

This call is only available from RISC OS 3.6 onwards.

Related SWIs

None

Related vectors

None

111 ADFS

Introduction and Overview

Logical block addressing (LBA)

Logical block addressing (or *LBA*) is a method of disc addressing for IDE discs, which is superseding the old *Cylinder-head-sector* (or *CHS*) method of disc addressing. The LBA method has been introduced by hard drive manufacturers because the CHS method would not work under MS-DOS for drives greater than 528 MB; so LBA is typically only used with these larger discs, and smaller discs continue to use CHS.

From RISC OS 3.6 onwards, ADFS supports IDE discs that use LBA. It recognises an LBA disc by a flag in the hardware dependant parameters of the boot block, which is set appropriately by formatting software such as HForm. The flag is at offset &1BA in the boot block; if bit 0 is set, the disc uses LBA.

The main advantage gained from the use of LBA is faster conversion of disc addresses. The disc address that FileCore passes to the low-level entry points of ADFS is a sector offset into the disc (which is the same as the LBA), with a drive number in the top bits. Converting to LBA just involves masking out the drive number, whereas converting to CHS requires two divisions by numbers which are only known at run-time. Since ADFS converts disc addresses during IRQ handling, using LBA improves IRQ latency.

Changes to existing SWIs

ADFS_DiscOp (page 2-283)

ADFS does not support reason code 3 for all hard discs. It also does not support bit 4 of the option bits (ie the ‘use alternate defect list’ bit).

New SWIs

Three new SWIs have been introduced in RISC OS 3.6:

- ADFS_LockIDE (page 5a-188) locks/unlocks the IDE bus.
- ADFS_SectorDiscOp (page 5a-187) calls FileCore_SectorOp; it hence provides the same functionality as ADFS_DiscOp, save that it uses sector addresses rather than byte addresses.
- ADFS_FreeSpace64 (page 5a-189) calls FileCore_FreeSpace64; it hence provides the same functionality as ADFS_FreeSpace, but uses 64 bit values rather than 32 bit ones.

SWI Calls

ADFS_SectorDiscOp (SWI &4024D)

Calls FileCore_SectorOp

On entry

See FileCore_SectorOp (page 5a-178)

On exit

See FileCore_SectorOp (page 5a-178)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_SectorOp (page 5a-178), after first setting R8 to point to the FileCore instantiation private word for ADFS.

ADFS does not support reason code 3 for all hard discs. It also does not support bit 4 of the option bits (ie the ‘use alternate defect list’ bit).

This call is functionally identical to FileCore_SectorOp.

Related SWIs

FileCore_SectorOp (page 5a-178), ADFS_DiscOp (page 2-283)

Related vectors

None

ADFS_LockIDE (SWI &40251)

Locks/unlocks the IDE bus

On entry

R0 = flags:
bit 0 clear ⇒ unlock IDE bus, set ⇒ lock IDE bus
all other bits reserved (must be zero)

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 locks/unlocks the IDE bus. An error is generated (&1080A, 'Driver in use') if the bus is already locked when you attempt to lock it.

When attempting to lock in the background, you should not attempt to loop, repeatedly locking, since the process in control of the lock could be a foreground process. Instead, you should be schedule a retry for a later time.

Related SWIs

None

Related vectors

None

ADFS_FreeSpace64 (SWI &40252)

Calls FileCore_FreeSpace64

On entry

See FileCore_FreeSpace64 (page 5a-183)

On exit

See FileCore_FreeSpace64 (page 5a-183)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI calls FileCore_FreeSpace64 (page 5a-183), after first setting R8 to point to the FileCore instantiation private word for ADFS.

This call is functionally identical to FileCore_FreeSpace64.

Related SWIs

FileCore_FreeSpace64 (page 5a-183), ADFS_FreeSpace (page 2-287)

Related vectors

None

112 DOSFS

Introduction and Overview

Support for larger DOSFS image files

Under RISC OS 3.5 and earlier a DOSFS image file had a maximum size of 32 MB. This limit was imposed by the DOS boot block used in laying out the image file.

RISC OS 3.6 uses a newer type of DOS boot block which removes this restriction.

Less stringent checking of DOS formats

Under RISC OS 3.6 and later, DOSFS is less stringent in its checking of DOS formats. Some discs that earlier versions of DOSFS rejected are now accepted. In particular DOSFS no longer checks for two FATs, and will accept discs that have just one FAT.

113 CDs and CD-ROMs

Introduction

Support has been added to RISC OS 3.6 for CDs and CD-ROMs. This software was previously separately available, and was typically supplied in a ROM on the SCSI card used to interface to a SCSI-based CD-ROM drive.

The software provides a filing system with which you can access files on a CD-ROM that conforms to the widely used ISO 9660 standard. It also provides commands with which you can play audio CDs, starting, stopping and pausing wherever you like. You can read audio data directly from a CD, provided the CD-ROM drive you are using supports this facility.

CDs can store about 75 minutes of audio data. As CD-ROMs, they can be used to store about 660 MB of data, making CDs suitable for mass data applications and as an affordable publishing medium.

The future of CD-related modules

Acorn intend replacing all CD-related components of RISC OS in the next release. Because of this, the SWI interface provided by the CDFS and CDFSDriver modules will become obsolete, and so we do not document it here. If you wish to write applications that use these SWIs, you should:

- **Contact Acorn Computers for details of the SWIs.**
- **Write your application so that code that calls the SWIs is separate from the rest of your application, and can be easily replaced in the future.**

We do document the * Commands provided by the CDFS module, solely so that you can use them from the command line. These may also become obsolete in the future, and you should treat them in the same way as SWIs when writing applications.

You will also need details of these SWIs if you wish to write a soft-loadable driver to support a new type of CD-ROM drive under the current CD system. Again, you should contact Acorn Computers for further details.

CDFS

The CDFS module is responsible for interpreting the data on a CD-ROM that uses the ISO 9660 standard, and ensuring the RISC OS filing system is properly supported.

CDFS is implemented as a FileSwitch-based filing system rather than as a FileCore module, because some aspects of CD-ROMs such as directory size, disc size and filename length can exceed limitations imposed by FileCore. Consequently you can use the standard FileSwitch SWIs – such as OS_Byte, OS_File, OS_Find and OS_GBPB – and * Commands to read files and data. Obviously you cannot write to disc, as CDs are not a writable medium; CDFS is a read-only filing system, and gives an error if you make a call that attempts to write data.

Because CDFS supports standard FileSwitch calls, correctly written applications will be able to use the CDFS filing system without modification.

This ‘standard FileSwitch interface’ will remain supported in the next release of RISC OS.

* Commands

*Bye

Ends a CDFS session

Syntax

```
*Bye
```

Parameters

None

Use

*Bye ends a CDFS session by closing all files, unsetting all directories and libraries, forgetting all CD-ROM names and parking the heads of CD-ROM drives to their ‘transit position’ so that they can be moved without risking damage to the read head.

You should check that CDFS is the current filing system before you use this command, or alternatively if another filing system is your current one, you can type:

```
*CDFS:Bye
```

Example

```
*Bye
```

Related commands

*Close (page 2-148), *Dismount (page 5a-201), *Shut (page 2-187), *Shutdown (page 2-188)

*CDDevices

*CDDevices displays all the CD devices connected, and information about them

Syntax

```
*CDDevices
```

Parameters

None

Use

*CDDevices displays all the CD devices connected, their product name, capacity, firmware revision, and their SCSI ID (displayed as device, LUN, and card; or as zeroes for non-SCSI devices). In more detail:

Drive	is the logical drive number assigned by CDFS
Dev, Lun and Card	are the device ID, logical unit and card numbers that together make up the drive's SCSI address
Product	is a brief identification of the CD-ROM drive provided by its manufacturer
Capacity	is the total amount of information (both data and audio) on the CD currently in the drive, or 'Unknown' if there is no readable CD in the drive
Firmware	is the version of the manufacturer's firmware fitted to the drive.

Unrecognised drive types are omitted from the list.

The information returned is liable to change in future versions; you should not rely on its content or format.

Example

```
*CDDevices
Drive  Dev  LUN  Card Product             Capacity      Firmware
00     0    0    0    CD-ROM CR-571       413 Mbytes    1.0e
```

Related commands

None

*CDFS

Selects the CD-ROM Filing System as the current filing system

Syntax

```
*CDFS
```

Parameters

None

Use

*CDFS selects the CD-ROM Filing System as the filing system for subsequent operations. Remember that it is not necessary to switch filing systems if you use the full pathnames of objects. For example, you can refer to NetFS objects (on a file server, say) when CDFS is the current filing system.

Example

```
*CDFS
```

Related commands

*ADFS (page 2-304), *Net (page 2-388), *RAM (page 2-322), *ResourceFS (page 2-425)

*CDSpeed

Displays or sets the read speed of a CD-ROM drive

Syntax

```
*CDSpeed [drive [speed]]
```

Parameters

drive	a valid CD-ROM drive number
speed	the new read speed for that drive: 1 standard speed 2 double speed 255 maximum speed

Use

*CDSpeed displays or sets the read speed of the given CD-ROM drive, or of the current drive if none is specified. To set the speed, you must specify both the drive number and the new speed.

Note that many drives only support a single read speed; if you attempt to set their read speed, you will get an ‘Invalid parameter’ error.

Example

```
*CDSpeed
Current speed setting is 2
```

Related commands

None

*Configure CDROMBuffers

Sets the configured amount of memory reserved for CD-ROM buffering

Syntax

```
*Configure CDROMBuffers size[K]
```

Parameters

size	the size of memory to reserve, in kilobytes: can be 0, 8, 16, 32, 64, 128, 256 or 512

Use

*Configure CDROMBuffers sets the configured amount of memory reserved for CD-ROM buffering, in kilobytes. This can be 0K, 8K, 16K, 32K, 64K, 128K, 256K, or 512K. If you specify any other size, then the next lowest value will be set.

The buffer space is used for a number of tasks such as cacheing blocks of data (ie 2048 bytes) and disc specific information – but it is mostly used to cache directory information. This saves accessing a directory and all its parent directories each time a request is made from it. With the slow seek time of CD-ROMs, this saving gives a significant performance increase, especially for deeply nested directories.

The performance of CDFS very much depends on it having adequate buffer space available. The desirable amount depends on various things: in particular, CDFS caches information for each disc in use, so using multiple CDs requires extra buffer space. Also, discs holding more objects have more directory information to cache. As a rough guide, a buffer size of 16 Kbytes is normally adequate for a single average CD.

To save memory usage, CDFS does not load at boot time if the computer is configured to have zero CD-ROM drives, and so this command will not be available. However, you can always use the Configure application to change all CDFS configuration settings, even if CDFS itself is not loaded.

Example

```
*Configure CDROMBuffers 64K
```

Related commands

None

*Configure CDROMDrives

Sets the configured number of CD-ROM drives recognised at power on

Syntax

`*Configure CDROMDrives` *n*

Parameters

n	the number of CD-ROM drives, in the range 0 - 27

Use

*Configure CDROMDrives sets the configured number of CD-ROM drives recognised at power on.

To save memory usage, CDFS does not load at boot time if the computer is configured to have zero CD-ROM drives, and so this command will not be available. However, you can always use the Configure application to change all CDFS configuration settings, even if CDFS itself is not loaded.

Example

```
*Configure CDROMDrives 1
```

Related commands

*Configure Floppies (page 2-308), *Configure HardDiscs (page 2-309), *Configure IDEDiscs (page 2-309)

*Dismount

Ensures that it is safe to finish using a CD-ROM

Syntax

```
*Dismount [disc_spec]
```

Parameters

disc_spec	the name of the CD-ROM or number of the CD-ROM drive

Use

*Dismount ensures that it is safe to finish using a CD-ROM by closing all its files, unsetting all its directories and libraries, forgetting its disc name, and parking its read head. If no CD-ROM is specified, the current CD-ROM is used as the default.
*Dismount is useful before removing a particular CD-ROM; however, the *Shutdown command is usually to be preferred, especially when switching off the computer.

Example

```
*Dismount
```

Related commands

*Mount (page 5a-205), *Shutdown (page 2-188)

*Drive

Sets the current CD-ROM drive

Syntax

`*Drive` *`drive`*

Parameters

`drive`	the number of the CD-ROM drive, from 0 - 27

Use

*Drive sets the current CD-ROM drive if NoDir is set. Otherwise, *Drive has no meaning.

Example

```
*Drive 3
```

Related commands

*Dir (page 2-163), *NoDir (page 2-175)

*Eject

Ejects the disc from a CD-ROM drive

Syntax

```
*Eject [drive]
```

Parameters

drive	a valid CD-ROM drive number

Use

*Eject ejects the disc from the given CD-ROM drive, or from the current drive if none is specified. This command will only work if the drawer has not been locked by the *Lock command (see page 5a-204), and is electronically operated.

Example

```
*Eject 0
```

Related commands

*Lock (page 5a-204), *Unlock (page 5a-211)

*Lock

Locks the disc in a CD-ROM drive, disabling the Eject button

Syntax

```
*Lock [drive]
```

Parameters

drive	a valid CD-ROM drive number

Use

*Lock locks the disc in the given CD-ROM drive, disabling the Eject button. If no drive is specified, the current drive is locked. You must call the *Unlock command (page 5a-211) before the disc can again be ejected.

Example

```
*Lock 0
```

Related commands

*Unlock (page 5a-211)

*Mount

Prepares a CD-ROM for general use

Syntax

`*Mount [`*`disc_spec`*`]`

Parameters

`disc_spec`	the name of the CD-ROM or number of the CD-ROM drive

Use

*Mount prepares a CD-ROM for general use by setting the current directory to its root directory, setting the library directory (if it is currently unset) to $.Library, and unsetting the User Root Directory (URD). If no disc spec is given, the default CD-ROM drive is used. The command is preserved for the sake of compatibility with earlier Acorn operating systems, and ideally you should not use it.

Example

```
*Mount :VIDEOCLIP2
```

Related commands

*Dismount (page 5a-201)

*Play

Plays from the specified audio track to the end of the disc in a CD_ROM drive

Syntax

```
*Play track [drive]
```

Parameters

track	track from which to start playing, in the range 0 - 99
drive	a valid CD-ROM drive number

Use

*Play plays from the specified audio track to the end of the disc in the given CD_ROM drive, or in the current drive if none is specified. No data is transferred to the computer; playback uses the drive's digital to analogue circuits and audio output – which is typically via a jack socket, phono sockets or other in-line adaptors.

If the track number does not exist on the CD in the drive, you will get the error 'Number too small' or 'Number too big'. If you try to play a data track, you will get the error 'Cannot play that data'.

Example

```
*Play 9 0
```

Related commands

*PlayMSF (page 5a-208), *Stop (page 5a-209)

*PlayList

Lists the tracks – whether audio or data – on the disc in a CD-ROM drive

Syntax

```
*PlayList [drive]
```

Parameters

drive	a valid CD-ROM drive number

Use

*PlayList lists the tracks – whether audio or data – on the disc in the given CD-ROM drive, together with their start time and the total CD time. If no drive is specified, the current disc's tracks are listed.

The start time is given as a 'Red Book address', in minutes, seconds, and frames (each of which is 1/75 of a second) from the start of the disc.

Example

```
*PlayList 0
Track number, contains, starts from MM:SS:FF
Track   01   is data    00:00:00
Track   02   is audio   23:24:65
Track   03   is audio   27:59:05
Total   03   track(s)   34:21:74
```

Related commands

None

*PlayMSF

Plays a piece of audio from the disc in a CD-ROM drive

Syntax

```
*PlayMSF mins:secs:frames mins:secs:frames [drive]
```

Parameters

mins	number of minutes from the start of the disc at which to start/stop playing
secs	number of seconds from the start of the disc at which to start/stop playing
frames	number of frames from the start of the disc at which to start/stop playing
drive	a valid CD-ROM drive number

Use

*PlayMSF plays a piece of audio from the disc in the given CD-ROM drive, or in the current drive if none is specified. The start and stop times time are specified as a 'Red Book address', in minutes, seconds, and frames (each of which is 1/75 of a second) from the start of the disc. The start time is the first of the two parameters.

Playing stops immediately a data track is encountered, so if the start time is in a data track this command will appear to do nothing. You will get an error if the start and/or end times lie outside the range of the CD.

Example

```
*PlayMSF 02:05:38 23:59:74
```

Related commands

*Play (page 5a-206), *Stop (page 5a-209)

*Stop

Stops playing the disc in a CD-ROM drive

Syntax

```
*Stop [drive]
```

Parameters

drive	a valid CD-ROM drive number

Use

*Stop stops playing the disc in the given CD-ROM drive, or in the current drive if none is specified. If the drive is not currently playing, this command is ignored.

Example

```
*Stop 0
```

Related commands

*Play (page 5a-206), *PlayMSF (page 5a-208)

*Supported

Lists the drive types recognised by CDFS

Syntax

```
*Supported
```

Parameters

None

Use

*Supported lists the drive types recognised by CDFS, and hence that are usable. The list only gives manufacturers' names, not model numbers.

RISC OS 3.6 nominally supports the following drives:

ATAPI	Conformant drives
Chinon	CDS-431
Hitachi	CDR-3650/1650S and CDR-1750S
Philips	CM212 and CDD521
Sony	CDU-6111, CDU-6211, CDU-541 and CDU-561
Toshiba	XM-3301 and XM-3401

However, since drives' firmware can change, you should not treat the above list as definitive. In particular, because the ATAPI standard is still in a state of flux, and not all drives conform to the standard anyway, you may find that not all so-called 'ATAPI' drives work with RISC OS 3.6. However, you may find some other drives made by the above manufacturers are sufficiently compatible to also work.

This call may not be supported in the future, or the information returned may change in content and/or format. You should therefore not use this call in applications or scripts.

Example

```
*Supported
SONY, LMS, TOSHIBA, HITACHI, CHINON      (LMS – Laser Magnetic
                                          Systems – is actually
Philips)
```

Related commands

None

*Unlock

Re-enables the Eject button on a CD-ROM drive

Syntax

```
*Unlock [drive]
```

Parameters

drive	a valid CD-ROM drive number

Use

*Unlock re-enables the Eject button on the given CD-ROM drive, reversing the effect of any earlier *Lock command. If no drive is specified, the current drive is unlocked.

Example

```
*Unlock 0
```

Related commands

*Eject (page 5a-203), *Lock (page 5a-204)

*WhichDisc

Displays the unique ID number for the disc in the current CD-ROM drive

Syntax

```
*WhichDisc
```

Parameters

None

Use

*WhichDisc displays the unique ID number for the disc in the current CD-ROM drive. The number is calculated from the information in the disc's TOC (as defined in the *Red Book*), therefore it is unlikely that two discs will have the same value.

Example

```
*WhichDisc
322279
```

Related commands

None

114 NetPrint

Introduction and Overview

For details of the NetPrint printing protocol, see *Printer server protocol interface* on page 5a-682.

A service call has been added to avoid potential clashes between Econet port numbers. It is described overleaf:

Service_NetPrintCheckD1 (Service Call &40200)

Issued by NetPrint to determine if there is a local printer server running

On entry

R1 = &40200 (reason code)

On exit

R1 = 0 to claim the command, or preserved to pass on

Use

This service call is issued by NetPrint to determine if there is a local printer server running.

If NetPrint is trying to print to a remote server, it should listen for replies on Econet ports &D0 and &D1, since old and new versions (respectively) of the printer server protocol use those ports. (See *Printer server protocol interface* on page 5a-682.) However, if a printer server is running on the local machine, it uses port &D1 to listen for data. Hence if both NetPrint and a local printer server are in use, packets can arrive at port &D1 for two different programs, with no way of telling the owner of a given packet.

To avoid any potential confusion, NetPrint issues this service call. If it is claimed there is a local printer server running, and so NetPrint can only listen on port &D0; it cannot communicate with printer servers that reply on port &D1 using the old protocol. If it is not claimed, NetPrint can listen on ports &D0 and &D1, and can communicate with older printer servers.

115 Parallel and serial device drivers

Introduction and Overview

This chapter outlines changes made in RISC OS 3.5 to the Buffer Manager, DeviceFS, and the serial and parallel device drivers in order to improve the performance of these ports.

Buffer Manager

The buffer manager has been extended to provide facilities for insertion and removal of buffered data without using SWI calls, hence avoiding all the related overheads. This is done by directly calling the *buffer manager service routine*, which uses a reason code to specify its action. The service routine provides all of the functionality of vectors InsV, RemV and CnpV, and has been based on the existing handlers in the buffer manager but optimised as much as possible. For full details of the various reason codes, see *The buffer manager service routine* on page 5a-217.

Device drivers wishing to use this service routine first have to call a new SWI – Buffer_InternalInfo, described on page 5a-228. This provides the information that is required to use the service routine with a particular buffer.

The existing vector interface is still supported, but takes the form of an extra layer on top of the new code.

DeviceFS module

The DeviceFS module has been modified to call the buffer manager service routine in all situations where InsV, RemV or CnpV were previously used; for example the calls DeviceFS_ReceivedCharacter and DeviceFS_TransmitCharacter in the filing system interface.

Parallel device driver

The parallel device driver has been modified to use the buffer manager service routine, hence greatly improving performance.

The parallel device can be opened either for input or output but not for both. When an input or output stream is created, the parallel device driver calls Buffer_InternalInfo (page 5a-228) to obtain the internal buffer ID for the relevant buffer and the address of the buffer manager service routine. All calls to InsV, RemV or CnpV have been replaced with calls to the buffer manager service routine.

Fast Centronics mode

The new I/O chips provide a fast Centronics mode where bytes written to the FIFO are automatically sent by the hardware at a very high transfer rate using STROBE and BUSY signals as the handshake. The parallel device driver accesses this mode using a new device called ‘fastparallel:’. To work with this device driver, technically speaking the printer must assert BUSY within 500ns of receiving STROBE; in practice, it should explicitly state it supports fast Centronics

The ‘parallel:’ device is still available as the default, since some printers cannot cope with the fast transfer rate of the new device.

Serial device driver

The serial device driver does not use the new buffer manager interface; this is to retain maximum compatibility with existing applications that use the serial interface.

However, its performance has been considerably improved, and it can now support a maximum serial port rate of 115200 baud. Other improvements have resulted in the elimination of most interrupt problems affecting serial input. At the maximum serial port rate of 115200 baud, the input FIFO will allow 1ms of interrupt latency before overrun occurs. This should be ample under most circumstances. The allowed latency increases as the baud rate is lowered.

OS_SerialOp has been extended with the addition of new baud rate codes and reason codes; see page 5a-224.

Technical Details

The buffer manager service routine

The buffer manager service routine provides direct access to buffers without the overheads of calling SWIs.

A device driver wishing to use the service routine should first create or register its buffers with the buffer manager. It must then call the SWI Buffer_InternalInfo (page 5a-228) for each buffer. This returns the address of the service routine and a pointer to its workspace (which are the same for all buffers), and an internal buffer ID specific to that buffer.

Calling the buffer manager service routine

The service routine provides various functions, specified by a reason code. It can be called in IRQ or SVC mode, interrupts may be enabled or disabled. Entry conditions are:

R0 = reason code (see below)
R1 = internal buffer ID
R12 = R2 value from Buffer_InternalInfo call
Other registers depend on reason code

Current reason codes are as follows:

R0	Action	Page
0	Insert byte	5a-218
1	Insert block	5a-218
2	Remove byte	5a-219
3	Remove block	5a-219
4	Examine byte	5a-220
5	Examine block	5a-220
6	Return used space	5a-221
7	Return free space	5a-221
8	Purge buffer	5a-221
9	Next filled block	5a-222

The service routine can use the internal buffer ID to go straight to the appropriate buffer record in the buffer manager’s workspace, rather than having to perform a linear search on a buffer handle.

On exit from the service routine, registers are normally preserved, save for those used to return results.

If the device driver removes or deregisters a buffer, it must ensure it no longer quotes that buffer's internal ID when calling the buffer manager service routine.

Insert byte

On entry

R0 = 0 (reason code)
R1 = internal buffer ID
R2 = byte to insert
R12 = R2 value from Buffer_InternalInfo call

On exit

All registers preserved

C = 1 $\Longrightarrow$ failed to insert

Use

This reason code inserts a byte into the specified buffer.

Insert block

On entry

R0 = 1 (reason code)
R1 = internal buffer ID
R2 = pointer to data to insert
R3 = number of bytes to insert
R12 = R2 value from Buffer_InternalInfo call

On exit

R2 = pointer to first byte not inserted
R3 = number of bytes not inserted
All other registers preserved

C = 1 $\Longrightarrow$ unable to transfer all data (ie. R3$\neq$0)

Use

This reason code inserts a block of data into the specified buffer. The pointer and length are adjusted to reflect how much data was actually inserted. If the data has already been written directly into the buffer (ie. R2 = pointer to buffer insertion point), then no data is copied and the buffer indices are simply updated.

Remove byte

On entry

R0 = 2 (reason code)
R1 = internal buffer ID
R12 = R2 value from Buffer_InternalInfo call

On exit

R2 = byte removed
All other registers preserved

C = 1 ⟹ unable to remove byte

Use

This reason code removes a byte from the specified buffer.

Remove block

On entry

R0 = 3 (reason code)
R1 = internal buffer ID
R2 = pointer to destination area
R3 = number of bytes to remove
R12 = R2 value from Buffer_InternalInfo call

On exit

R2 = pointer to first free byte in destination area
R3 = number of bytes not removed
All other registers preserved

C = 1 ⟹ unable to remove all data (ie. R3≠0)

Use

This reason code removes a block from the specified buffer. The pointer and length are adjusted to reflect how much data was actually removed.

Examine byte

On entry

R0 = 4 (reason code)
R1 = internal buffer ID
R12 = R2 value from Buffer_InternalInfo call

On exit

R2 = next byte to be removed
All other registers preserved

C = 1 ⟹ unable to get byte

Use

This reason code reads the next byte to be removed from the specified buffer, without actually removing it.

Examine block

On entry

R0 = 5 (reason code)
R1 = internal buffer ID
R2 = pointer to destination area
R3 = number of bytes to examine
R12 = R2 value from Buffer_InternalInfo call

On exit

R2 = pointer to first free byte in destination area
R3 = number of bytes not transferred
All other registers preserved

C = 1 ⟹ unable to transfer all data (ie. R3≠0)

Use

This reason code reads a block of data from the specified buffer, without actually removing it. The pointer and length are adjusted to reflect the data transferred.

Return used space

On entry

R0 = 6 (reason code)
R1 = internal buffer ID
R12 = R2 value from Buffer_InternalInfo call

On exit

R2 = number of used bytes in buffer
All other registers preserved

Use

This reason code returns the number of bytes in the specified buffer.

Return free space

On entry

R0 = 7 (reason code)
R1 = internal buffer ID
R12 = R2 value from Buffer_InternalInfo call

On exit

R2 = number of free bytes in buffer
All other registers preserved

Use

This reason code returns the number of free bytes in the specified buffer.

Purge buffer

On entry

R0 = 8 (reason code)
R1 = internal buffer ID
R12 = R2 value from Buffer_InternalInfo call

On exit

All registers preserved

Use

This reason code purges all data from the specified buffer.

Next filled block

On entry

R0 = 9 (reason code)
R1 = internal buffer ID
R3 = number of bytes read since last call
R12 = R2 value from Buffer_InternalInfo call

On exit

R2 = pointer to first byte in next block to be removed
R3 = number of bytes in next block
All other registers preserved

C = 1 $\Longrightarrow$ buffer empty

Use

This reason code can be used to remove buffered data directly, rather than copying it from the buffer using reason code 3. Initially, the call should be made with R3 = 0 so that no bytes are purged. The call returns a pointer to the next byte to be removed from the buffer, and the number of bytes which can be removed from that address onwards. In the next call R3 should equal the number of bytes read since the last call, at which point the buffer indices will be updated to purge the data, and the next filled block will be returned.

A device driver which uses this call must be the only application which removes data from the buffer.

Changes to existing SWIs

OS_ReadSysInfo (page 1-746)

This call has been extended to provide information on the new hardware supported by RISC OS 3.5.

- Extra values are now returned by reason code 2 (read presence of chips and unique machine ID) to allow support of new hardware features. This is described below.
- The values returned by reason code 3 (read features mask) differ for the new hardware, although the call itself has not changed, Again, this is described below.

Furthermore, two new reason codes (4 and 5) have been added. However, these are for internal use only, and you must not use them in your own code.

OS_ReadSysInfo 2 (page 1-750)

This reason code has been extended in a backward-compatible manner to return information on the new hardware supported by RISC OS 3.5. This has been done by splitting into fields the values returned in R0 - R2 on exit:

R0 = hardware configuration word 0:
- bits 0 - 7 = special functions chip type:
  - 0 ⟹ none, 1 ⟹ IOEB ASIC
- bits 8 - 15 = I/O control chip type:
  - 0 ⟹ IOC, 1 ⟹ IOMD
- bits 16 - 23 = memory control chip:
  - 0 ⟹ MEMC1/MEMC1a, 1 ⟹ IOMD
- bits 24 - 31 = video control chip type:
  - 0 ⟹ VIDC1a, 1 ⟹ VIDC20

R1 = hardware configuration word 1:
- bits 0 - 7 = I/O chip type:
  - 0 ⟹ absent, 1 ⟹ 82C710/711 or SMC ’665 or similar
- bits 8 - 31 reserved (set to 0)

R2 = hardware configuration word 2:
- bits 0 - 7 = LCD controller type:
  - 0 ⟹ absent, 1 ⟹ present (type 1)
- bits 8 - 31 reserved (set to 0)

The unique machine ID is still returned in R3 and R4, if available.

OS_ReadSysInfo3 (page 1-751)

This reason code has not been altered in functionality. However the values returned in R0 and R1 have altered, because RISC OS 3.5 machines do not use the 710/711 family of chips, but instead use the broadly compatible SMC ’665 family. Values returned in R0 are:

R0 bits	sub-unit	SMC ’665
0 - 3	IDE hard disc interface	1
4 - 7	floppy disc interface	1
8 - 11	parallel port	1
12 - 15	1st serial port	1
16 - 19	2nd serial port	1
20 - 23	chip configuration	3
24- 31	reserved	0

The only difference is the chip configuration, since the sub-units described still have the same basic functionality. The SMC ’665 has extra functionality: you can use a fast parallel mode (with FIFO and hardware handshake), and use the serial FIFOs provided. Hence the extra features mask returned in R1 differs to reflect this:

R1 bits	sub-unit	SMC ’665
0 - 3	IDE hard disc interface	0
4 - 7	floppy disc interface	0
8 - 11	parallel port	1
12 - 15	1st serial port	1
16 - 19	2nd serial port	1
20 - 23	chip configuration	0
24- 31	reserved	0

OS_SerialOp (page 2-468)

New flag bit

OS_SerialOp 0 (page 2-470) accepts a new flag bit to enable or disable the serial FIFOs (if present):

Bit	Read/Write or Read Only	Value	Meaning
8	R/W	0	Disable the serial FIFOs, if present
		1	Enable the serial FIFOs, if present

New baud rates

OS_SerialOp 5 and 6 (page 2-479 and page 2-481) accept new baud rate codes to support the higher baud rates possible under RISC OS 3.5. These are:

Value of R1	Baud rate
16	38400
17	57600
18	115200

New reason code

This call has also been extended by the addition of a new reason code, described later in this chapter:

- OS_SerialOp 9 (page 5a-226) enumerates the available serial port speeds.

OS_Byte 7 (page 2-451) and 8 (page 2-453)

These calls have been updated to support the new reason codes used by OS_SerialOp 5 and 6 (see above). However, as in RISC OS 3, you should use the OS_SerialOp calls in preference.

SWI calls

OS_SerialOp 9 (SWI &57)

Enumerates the available serial port speeds

On entry

R0 = 9 (reason code)

On exit

R0 preserved
R1 = pointer to table of supported baud rates
R2 = number of entries in table

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 enumerates the available serial port speeds, returning them as a table. The table is word aligned; each word in the table specifies a baud rate in units of 0.5 bit/sec. (This is to support rates such as 134.5 baud.)

The index into the table (starting at 1) can be used in OS_SerialOp 5 and 6 calls to set the corresponding baud rate.

This call is available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

SerialV

Buffer_InternalInfo (SWI &42949)

Converts a buffer handle to a buffer manager internal buffer ID

On entry

R0 = buffer handle

On exit

R0 = internal buffer ID
R1 = address of buffer manager service routine
R2 = value to pass to service routine in R12

Interrupts

Interrupt status is not altered
Fast interrupts are not altered

Processor mode

Processor is in SVC mode.

Re-entrancy

SWI is not re-entrant

Use

This call converts the buffer handle passed in R0 to a buffer manager internal buffer ID, which is specific to that buffer. It also returns the address of the buffer manager service routine (see page 5a-217), and the value to quote in R12 when calling the service routine; these are the same for all buffers.

If the buffer handle is invalid an error is returned, but can be ignored; the service routine address and R12 value will still be returned.

This call is available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

116 Keyboard and mouse

Introduction and Overview

One of the main changes in RISC OS 3.5 was the removal of the Acorn keyboard interface from the kernel and its replacement with a standard IBM PS/2 compatible keyboard device driver, held in the separate ‘Keyboard’ module. For a description, see *The keyboard interface* on page 5a-232.

The standard quadrature mouse driver was also removed form the kernel and is now a separate driver, held in the ‘Mouse’ module. There is also a serial mouse driver that can be used if you connect a standard PC-type (Microsoft or Mouse Systems) mouse to the serial port; this is held in the ‘SerialMouse’ module.

In RISC OS 3.6 the PS/2 keyboard module was renamed ‘PS2Driver’, and extended to support an IBM PS/2 compatible mouse.

For more details of mouse drivers see *The pointer interface* on page 5a-234.

Technical details

The keyboard interface

The keyboard interface has been changed to remove hardware dependent code from the kernel to separate *keyboard device driver* modules. This makes it easier to support different keyboard devices, and hence open up the choice of keyboards that can be connected. RISC OS 3.5 and 3.6 both supply one keyboard device driver, suitable for IBM PS/2 compatible keyboards.

The interface allows more than one keyboard device to provide input at any one time. Input from multiple devices is merged into one stream as if coming from one device.

The keyboard device driver and kernel communicate with each other through the KeyV software vector (page 5a-240). The communication is two-way, so both the driver and the kernel need to claim the vector using OS_Claim (page 1-66), and install a routine to handle the calls that the other may make.

(KeyV is a software vector that was used in 8 bit Acorn machines, but has not been used since, and has now been redefined.)

The IOMD chip

The keyboard is connected to the new IOMD chip, or to the I/O circuitry integrated into an ARM 7500 or similar, rather than to the IOC chip used in earlier versions of RISC OS. The PS/2 compatible interface provided is similar to that provided by IOC, and includes:

- interrupts on receiver full
- interrupts on transmitter empty
- independent transmit and receive data registers
- automatic parity generation on transmitted data
- status and control line registers, capable of driving the keyboard Clock and Data lines.

The keyboard device driver

The keyboard device driver claims the keyboard device interrupts that IOMD generates by calling OS_ClaimDeviceVector (page 1-123). It assumes that the keyboard sends scan codes from IBM-MF compatible code set 2 (standard PS/2 code set); if the keyboard does not, then you may get unexpected results.

The keyboard device driver converts the scan codes to the low level key numbers expected by the RISC OS kernel. It then passes these to the kernel by calling the KeyV software vector whenever a key is pressed or released; a reason code indicates which has occurred. The keyboard device driver keeps a table of flags for key states, and only calls KeyV when the state changes.

The keyboard firmware's own auto-repeat capability is not used. Keys are instead repeated by the kernel – just as in earlier versions of RISC OS – hence keeping the same scheme for configuring auto-repeat delay and repeat rate.

The supplied keyboard device driver(s) can be replaced by a custom version if required (eg for a special needs input device). If you wish to use some other device with this vector, contact Acorn Technical Support for a keyboard ID allocation.

The keyboard handler

The keyboard handler is similar to that in earlier versions of RISC OS. It consists of a look up table and a small amount of code. It converts low-level key numbers provided by the keyboard device driver into an ASCII form, with extensions for special characters.

The keyboard handler can be replaced by a custom version if required (eg to support a foreign keyboard).

The kernel keyboard driver

The kernel keyboard driver is a part of the RISC OS kernel, and binds together the keyboard device driver and keyboard handler. The kernel keyboard driver uses the keyboard handler to convert the low-level key numbers into a recognisable form. The kernel keyboard driver also debounces key presses, keeps track of keys down, and generates auto-repeats of keys at the configured rate.

The kernel keyboard driver also tracks the state of the keyboard's LEDs, and calls KeyV to inform the keyboard device driver when it needs to change the state of an LED.

Supporting different keyboards

Now that different versions of RISC OS support different keyboards, you will find that the labels on key tops differ for certain key codes sent to applications. The *RISC OS 3 Style Guide* specifies applications' behaviour in terms of these labels, rather than key codes. Consequently, your application's behaviour (and possibly its help text and keyboard shortcuts) will have to change depending on which keyboard is in use. You can find this out by calling OS_InstallKeyHandler (page 1-945) with R0 = 1.

The pointer interface

Just as for the keyboard interface, all hardware dependent code has been removed from the kernel and placed in separate *pointer device driver* modules. Multiple pointing devices can exist on the computer, but only one can be active at any one time.

RISC OS 3.5 supplies two pointer device drivers:

- The first is suitable for a quadrature mouse (such as has been used on all previous versions of RISC OS). The IOMD chip provides a quadrature interface, and so machines fitted with IOMD normally use this driver.
- The second drives a PC serial mouse that uses either Microsoft or Mouse Systems data formats. It is provided so that users can choose an alternative pointer device from the large range available that uses these data formats.

These two drivers are held in separate modules.

RISC OS 3.6 adds a further pointer device driver:

- This driver is for a serial mouse that uses PS/2 data formats. The ARM 7500 provides two PS/2 interfaces (one for the keyboard, one for the mouse), but does not provide a quadrature interface, and so machines fitted with an ARM 7500 or similar normally use this driver.

This driver is held in the same module as the PS/2 keyboard device driver

Again a vector is used to communicate between the kernel and the device driver; both need to claim the vector and install a routine to handle the calls the other may make. The vector is a new one, named PointerV (page 5a-242). The interface it provides is common to all pointer device drivers. However, the drivers obviously differ in the way that they access the pointer device's hardware.

Passing the pointer position to the kernel

The kernel requests pointer device movements every VSync by calling PointerV with reason code 0; the pointer device driver returns the movements. The kernel then scales the pointer device movements depending on the configured mouse step, and updates the pointer position on the display.

The kernel is also responsible for:

- registering the pointer device buffer with the buffer manager
- all pointer device bounding
- responding to OS_Mouse calls.

Passing button presses to the kernel

When any buttons on the pointer device change state, the pointer device driver passes this to the kernel by calling KeyV, just as for a keyboard. The kernel treats these in the same way as any other key, including debouncing them.

Pointer device types

Most calls use a pointer device type to differentiate between the supported pointer devices. Currently defined devices are:

Type	Device
0	Quadrature mouse
1	Microsoft mouse
2	Mouse Systems mouse
3	PS/2 mouse (RISC OS 3.6 onwards)

If you wish to use some other device with this vector, contact Acorn Technical Support for a pointer device type allocation.

Configuring and selecting the pointer device type

The new command *Configure MouseType (page 5a-245) configures the pointer device type to use thereafter. A new SWI, OS_Pointer (page 5a-244), sets or gets the currently selected pointer device type. If a new type is selected, the kernel calls PointerV with reason code 2 so that pointer device drivers can enable or disable.

A further PointerV reason code of 1 can be used to enumerate the available pointer devices as text. This has been incorporated in the Configure application, so users can configure pointing devices from a menu.

Initialising a pointer device driver

When a pointer device driver initialises it must check the current pointer device type using OS_Pointer; should the driver understand the type, it must enable itself.

The quadrature mouse driver

A quadrature mouse is connected to IOMD, which does not provide interrupt support for mouse input. Instead it provides two 16-bit registers (for x and y directions) which increment, decrement and wrap when the mouse is moved. The state of the mouse buttons is stored in a specific memory location.

The quadrature mouse driver responds to requests for pointer device type 0. It polls the mouse position registers in IOMD, and calculates the mouse movements to return to the kernel by comparing the previous values of these registers with the new ones. Like wise, it regularly reads the state of the buttons.

The Microsoft / Mouse Systems serial mouse driver

The Microsoft / Mouse Systems serial mouse driver responds to requests for pointer device type 1 (Microsoft) or 2 (Mouse Systems). When it is selected (see *Configuring and selecting the pointer device type* on page 5a-235 and *PointerV* on page 5a-242), it configures the serial device using OS_SerialOp (page 2-468) and opens the `serial:` device for input.

The driver also claims TickerV (page 1-99), and processes any data received by the serial device on centisecond clock ticks. The code re-enables interrupts to avoid any adverse effects on interrupt latency, and sets a flag to prevent re-entrancy while it is being executed. All mouse movements are amalgamated until the kernel calls PointerV to request they be sent.

The driver does not prevent the reconfiguration of the serial port while it is active itself; however it ensures that the `serial:` device is reopened if it is closed by an external source. If another pointer device becomes selected, the driver releases TickerV and closes the `serial:` device.

The PS/2 serial mouse driver

The PS/2 serial mouse driver calls PointerV in a similar way to the Microsoft / Mouse Systems serial mouse driver; see *The Microsoft / Mouse Systems serial mouse driver* above.

However, since it uses its own dedicated interface, it does not claim TickerV, nor does it configure and use the serial device.

Data formats for serial mice

The various serial mouse drivers communicate with serial mice which transmit data in one of these formats:

Microsoft

The first class of mice are Microsoft compatible, and are supported from RISC OS 3.5 onwards. They send data reports in the following format:

	Bit 6	5	4	3	2	1	0
Byte 1	1	L	R	Y7	Y6	X7	X6
2	0	X5	X4	X3	X2	X1	X0
3	0	Y5	Y4	Y3	Y2	Y1	Y0
4	0	M	DT4	DT3	DT2	DT1	DT0

L, R, M	bit flags for left, right and middle buttons: 1 ⇒ button down
X7 - X0	signed x distance, in range –128 (left) to +127 (right)
Y7 - Y0	signed y distance, in range –128 (up) to +127 (down)
DT4 - DT0	device type: 0 ⇒ mouse, all others reserved

Some three button mice omit the 4th byte in the report, and set both the L and R bits when the middle button is pressed. The driver can cope with this, and still detects the state of the middle button. This feature is **not** intended to support 2 button mice.

Mouse Systems

The second class of mice are Mouse Systems Corporation compatible, and are supported from RISC OS 3.5 onwards. They send their data reports in this format.

	Bit 7	6	5	4	3	2	1	0
Byte 1	1	0	0	0	0	L	M	R
2	X7	X6	X5	X4	X3	X2	X1	X0
3	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
4	X7	X6	X5	X4	X3	X2	X1	X0
5	Y7	Y6	Y5	Y4	Y3	Y2	Y2	Y0

L, R, M	bit flags for left, right and middle buttons: 0 ⇒ button down
X7 - X0	signed x distance, in range –128 (left) to +127 (right)
Y7 - Y0	signed y distance, in range –128 (down) to +127 (up)

The second set of X, Y data (bytes 4 and 5) is not a duplicate of the first, but the movement of the mouse during transmission of the first report. It cannot be discarded.

PS/2

The third class of mice are PS/2 compatible, and are supported from RISC OS 3.6 onwards. They send their data reports in this format.

	Bit 7	6	5	4	3	2	1	0
Byte 1	Yv	Xv	Y8	X8	1	M	R	L
2	X7	X6	X5	X4	X3	X2	X1	X0
3	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

L, R, M	bit flags for left, right and middle buttons: 1 ⇒ button down
X8 - X0	signed x distance, in range –256 (left) to +255 (right)
Y8 - Y0	signed y distance, in range –256 (down) to +255 (up)
Xv	x data overflow: 1 ⇒ overflow
Yv	y data overflow: 1 ⇒ overflow

Protocols for serial mice

The Microsoft / Mouse Systems serial mouse driver

This driver only accepts data from mice communicating in a stream mode operating at 1200 baud. It does not support the higher baud rates that some mice allow you to select by sending them a command.

In Microsoft compatible format data is transferred in 7-bit bytes framed with 1 start bit and 2 stop bits with no parity.

In Mouse Systems Corporation compatible format data is transferred in 8-bit bytes framed with 1 start bit and 2 stop bits with no parity.

The PS/2 serial mouse driver

The PS/2 serial device driver uses an asynchronous serial port.

In PS/2 compatible format data is transferred in 8-bit bytes, preceded by 1 start bit, and followed by an odd parity bit and a stop bit.

Software vectors

KeyV
(Vector &13)

Used to communicate between the kernel and a keyboard device driver

On entry

Register usage is dependent on a reason code held in R0:

Keyboard present

R0 = 0 (reason code)
R1 = keyboard ID: 1 ⇒ Archimedes keyboard, 2 ⇒ PC-AT keyboard

Key released

R0 = 1 (reason code)
R1 = low-level internal key number (see *Low-level internal key numbers* on page 1-158)

Key pressed

R0 = 2 (reason code)
R1 = low-level internal key number (see *Low-level internal key numbers* on page 1-158)

Notify driver of LED state

R0 = 3 (reason code)
R1 = LED status flags:

Bit	Meaning when set
0	Scroll Lock on
1	Num Lock on
2	Caps Lock on
3 - 31	reserved (should be ignored)

Enable keyboard device drivers

R0 = 4 (reason code)

Reserved for Acorn use

R0 = 5 - 10 (reason codes)

On exit

All registers preserved

Use

All of these calls should be passed on; none of them should normally be intercepted.

Keyboard present

When a keyboard device driver has successfully initialised, it must notify the kernel that the keyboard is present by calling KeyV with this reason code.

Key released and Key pressed

When a key is released or pressed, a keyboard device driver must inform the kernel by calling KeyV with these reason codes. It must not do so until it is enabled by the kernel; see *Enable keyboard device drivers* below.

The key numbers are the same as those used by the key up/down event; see *Low-level internal key numbers* on page 1-158.

Notify driver of LED state

When the state of the keyboard LEDs changes, the kernel calls KeyV with this reason code. A keyboard device driver must claim KeyV, and install a routine to handle such calls by setting the keyboard's LEDs as requested.

Enable keyboard device drivers

The kernel calls KeyV with this reason code to enable keyboard device drivers. A keyboard device driver must not use the Key released and key pressed reason codes until it has received this call; any attempt to do so will be ignored.

This is **not** a reset call, and keyboard device drivers may see this call many times while they are active. However, it does mean that the kernel has flushed its table of keys that are held down, so the device driver should do the same if appropriate.

Reserved for Acorn use

Reason codes 5 - 10 are reserved for Acorn use.

Related SWIs

OS_Claim (page 1-66)

PointerV (Vector &26)

Used to communicate between the kernel and a pointer device driver

On entry

Register usage is dependent on a reason code held in R0:

Request status of pointer device

R0 = 0 (reason code)
R1 = device type – see page 5a-235

Enumerate pointer device types

R0 = 1 (reason code)
R1 = pointer to previously found driver's device type record list, or 0 if none

Pointer device type selected

R0 = 2 (reason code)
R1 = device type

On exit

All registers preserved except:

Request status of pointer device

R2 = signed 32-bit x movement since last call
R3 = signed 32-bit y movement since last call

Enumerate pointer device types

R1 = pointer to driver's device type record list

Use

Request status of pointer device

The kernel calls PointerV with this reason code every VSync, to obtain the latest movement of the pointer device. A pointer device driver that supports the specified device type should intercept the call, returning the movement of the pointing device since the last time this reason code was called. Otherwise it should pass the call on.

The kernel uses the returned values to update the pointer position.

Enumerate pointer devices

The kernel calls PointerV with this reason code to enumerate the available pointer device types. A pointer device driver must claim PointerV, and install a routine that adds to a linked list of pointer devices. It must add one record for each device type it supports:

Offset	Meaning
0	*next pointer*, giving address of next record
4	flags: bits 0 to 31 reserved (must be 0)
8	device type – see page 5a-235
9	name of pointer device, no more than 30 characters, null terminated (for use in menus)

The pointer device driver must claim the space for the records from RMA. It must set the *next pointer* field of the last record it added to the value that R1 had on entry, and pass on the call with R1 pointing to the first record it added.

The caller must later free the memory claimed from RMA, usually as it reads the returned list.

This call must not be intercepted.

Pointer device type selected

The kernel calls PointerV with this reason code when a device type is selected by OS_Pointer (see page 5a-244). A pointer device driver should enable itself if it supports the specified device type; otherwise it should disable itself.

This call must not be intercepted.

SWI calls

OS_Pointer (SWI &64)

Gets or sets the currently selected pointer device type

On entry

R0 = reason code: 0 ⇒ get pointer type, 1 ⇒ set pointer type
R1 = pointer device type (if R1 = 1 on entry) – see page 5a-235

On exit

R0 = pointer device type (if R1 = 0 on entry) – see page 5a-235

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call gets or sets the currently selected pointer device type. This is used to ensure that the correct pointer device driver responds to certain PointerV calls.

Selecting a new device type causes PointerV (page 5a-242) to be called with reason code 2 (Pointer device type selected), so that drivers can enable or disable.

Related SWIs

None

Related vectors

PointerV (page 5a-242)

* Commands

*Configure MouseType

Sets the configured pointer device to be used thereafter

Syntax

```
*Configure MouseType device_type
```

Parameters

device_type	a number giving the pointer device type (see page 5a-235)

Use

*Configure MouseType sets the configured pointer device to be used thereafter.

Example

```
*Configure MouseType 0          Select Quadrature mouse
```

117 Filing system locking and resets

Introduction and Overview

The FSLock module (added in RISC OS 3.5) provides protection against inadvertent or malicious changing of the CMOS RAM and hard disc contents. To do this it intercepts the calls that update the contents of the hard disc and CMOS RAM, and returns an error instead.

The Reset behaviour has been further changed in two ways. Firstly, it has been simplified both to reduce the confusing range of options that were available in earlier versions of RISC OS and to ensure a reset always really starts the machine afresh. Secondly, a link can be set on the machine's circuit board to prevent resetting the CMOS RAM by a Delete power-on or similar combination.

Technical Details

Changes to power-on and reset

Hardware CMOS protection

Earlier versions of RISC OS allowed users to reset some or all of CMOS RAM by holding down various keys whilst the machine was powered on. However, any resultant accidental or deliberate alteration of CMOS RAM could be a nuisance in some environments. To counteract this, RISC OS 3.5 has added support for a CMOS protection connector inside the machine.

With the connector in the protected position, the CMOS RAM cannot be cleared as a part of a power-on or reset sequence, no matter what keys are held down. With it in the unprotected position, CMOS RAM clearing works just as in earlier versions of RISC OS.

New reset behaviour

The power-on and reset combinations for RISC OS 3.5 have been changed to rationalise a previously confusing set of options.

Under earlier versions of RISC OS a reset had many variations depending on whether it was a power-on reset, ordinary reset or break style reset, whether *FX200,2 had been done before the reset, and so on. To most users this degree of flexibility was never useful simply because it was so complex.

Under RISC OS 3.5 the hardware generates the same type of reset at power-on and when the reset button is pressed. Both are now effectively hard resets; the previous concept of hard and soft resets is no more. In both cases RISC OS 3.5 goes through the full sequence of reset operations. It:

- Performs a self test
- Clears RAM
- Checks the keyboard for CMOS RAM clearing
- Initialises the OS.

You can still use the Break key as part of a reset combination (see below). This performs a partial reset that omits the self test and CMOS RAM clearing.

The following scheme is now used.

Key combination	Function
Power-on	Normal reset, use boot options
Ctrl-break	Partial reset (no self-test or CMOS RAM reset), use boot options
Reset	Normal reset, use boot options

The following modifiers can be used in conjunction with the above resets to change the boot behaviour:

Modifier	Function
Shift	Reverse action of configured boot option
* (on keypad)	Use boot options, but boot to command line instead of the configured language

For backward compatibility, pressing Shift-Break causes the same action as Shift-Ctrl-Break.

The following modifiers can be used to reset some or all of CMOS RAM, provided the CMOS protection connector is in the unprotected position:

Modifier	Function
Delete	Reset CMOS RAM
R	Partially reset CMOS RAM
Copy	As Delete, but configures separate sync
T	As R, but configures separate sync
0 to 9 (on keypad)	Configures monitor type
. (on keypad)	Configure auto monitor type, sync and mode

The FSLock Module

The FSLock module protects the CMOS RAM and hard disc against unwanted modification. It does so by intercepting any SWIs that alter the hard disc contents or CMOS RAM, and returning an error instead.

FSLock cannot protect all discs on all filing systems; it can only protect drives 4 - 7 on any one filing system. By default, the Configure application sets FSLock to protect the ADFS hard discs 4-7.

Of course, a machine which allows no hard disc updates is not very useful, so two areas of a protected disc have been left unprotected:

- $.Public can be used for general file storage; it cannot be created while the computer is in a locked state.
- $.!Boot.Resources.!Scrap.ScrapDir is writable to allow Scrap transfers of files between applications.

Lock states

FSLock operates in three states:

Fully unlocked

A fully unlocked machine has no password allocated to it, and can have its hard discs or configuration modified. This state persists over any sort of reset, and is the default selected after the CMOS RAM has been successfully cleared.

Partially unlocked

A partially unlocked machine has a password allocated to it, but can still have its hard discs and configuration modified. If reset the machine reverts to being locked.

Locked

A locked machine has a password allocated to it, and cannot have its hard discs or configuration modified. The machine stays in this state if it is reset.

Lock status

The lock states are passed to commands using a lock status:

Status	Meaning
0	Fully unlocked
1	Partially unlocked
2	Locked

Permitted passwords

The password is case sensitive. The Configure application restricts the password to at least five non-space characters that are acceptable in a writable icon. Although the SWIs will accept any null terminated string, we strongly recommend you stay within the restrictions imposed by the Configure application's interface, otherwise users may find the machine locked with an untypable password.

Changes to existing SWIs

OS_Byte 253 (page 1-935)

Under RISC OS 3.5 and later this call will always read the last type of reset as a power-on reset (R1 = 1 on exit).

SWI calls

FSLock_Version (SWI &44780)

Returns information describing the FSLock module

On entry

—

On exit

R0 = version number × 100
R1 = pointer to module's workspace

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 returns information describing the FSLock module. R0 gives the module's version number, and R1 gives a pointer to the module's workspace.

This call is available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

FSLock_Status (SWI &44781)

Returns the current lock status, and the locked filing system's number

On entry

—

On exit

R0 = lock status (page 5a-250)
R1 = locked filing system number (undefined if lock status = 0)

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 returns the current lock status, and the locked filing system's number.

This SWI can only be called by number; not by name.

This call is available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

FSLock_ChangeStatus (SWI &44782)

Changes one or more of the lock status, the password and the locked filing system

On entry

R0 = new lock status (page 5a-250)
R1 = pointer to current file locking password
R2 = pointer to new file locking password
R3 = new locked filing system number

On exit

R0 - R3 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 changes one or more of the lock status, the password and the locked filing system. The new lock status must always be passed in R0; other parameters may be required depending on its value, and the current lock status:

		Current lock status		
		0	**1**	**2**
New lock status	**0**	—	R1	R1
	1	R2, R3	R1, R2, R3	R1
	2	R2, R3	—	R1, R2, R3

If the old password is needed and not given correctly an error will be returned. If a filing system number is needed then a check will be made for that filing system's existence.

This SWI can only be called by number; not by name.

This call is available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

* Commands

*FSLock_ChangePassword

Changes the locked filing system and password

Syntax

```
*FSLock_ChangePassword fs_name [new_password [new_password [old_password]]]
```

Parameters

fs_name	a filing system name
new_password	new file locking password
old_password	current file locking password

Use

*FSLock_ChangePassword changes the locked filing system and password. If the machine was fully unlocked then the old password need not be given. If any of the passwords aren't given then a prompt appears where the password can be entered without it being seen, since each character typed is displayed on the screen as a hyphen ('-').

This command is available from RISC OS 3.5 onwards.

Example

```
*FSLock_ChangePassword scsifs
New password: ------
New password again: ------
Old password: --------
```

Related commands

None

*FSLock_Lock

Locks the computer from the partially unlocked state

Syntax

```
*FSLock_Lock
```

Use

*FSLock_Lock locks the machine from the partially unlocked state.

If the machine is fully unlocked or locked then an error message is given.

This command is available from RISC OS 3.5 onwards.

Related commands

None

*FSLock_Status

Displays the machine's current lock state

Syntax

```
FSLock_Status
```

Use

*FSLock_Status displays the machine's current lock state.

This command is available from RISC OS 3.5 onwards.

Example

```
*FSLock_Status
No filing system is currently locked
```

Related commands

None

*FSLock_Unlock

Unlocks the computer

Syntax

```
*FSLock_Unlock [-full] [password]
```

Parameters

password	current file locking password

Use

*FSLock_Unlock unlocks the computer.

If the `-full` switch is given then the machine will be fully unlocked, otherwise a partial unlock will be done. If the password isn't given then a prompt appears where the password can be entered without it being seen, since each character typed is displayed on the screen as a hyphen ('-').

If the machine is already in the required state (partially or fully unlocked) then an appropriate error will be given.

This command is available from RISC OS 3.5 onwards.

Example

```
*FSLock_Unlock -full gOL9pGbH
```

Related commands

None

118 Free

Introduction and Overview

The Free module has been updated in RISC OS 3.6 to support displaying free space in the desktop for filing systems with discs of more than 4 GB capacity.

Technical details

Changes to existing SWIs

Free_Register (page 2-522)

If a filing system's free space routine does not recognise a reason code passed to it, it should return with all registers preserved.

A new reason code has been defined in RISC OS 3.6 for the filing system's free space routine (as registered using this call). The new reason code (4) returns the free space on the disc in 64 bits, rather than the 32 bits allowed by reason code 2:

Reason code 4– Get 64 bit free space for device

On entry

R0 = 4
R1 = filing system number
R2 = pointer to 6 word buffer
R3 = pointer to device name / ID

On exit

R0 = 0
R1 - R3 preserved

Details

This entry point is called to get the free space for a device. You should fill in the buffer pointed to by R2 with the following information:

Offset	Meaning
0	bits 0 - 31 of total size of device (0 if unchanged from last time read)
4	bits 32 - 63 of total size of device (0 if unchanged from last time read)
8	bits 0 - 31 of free space on device
12	bits 32 - 63 of free space on device
16	bits 0 - 31 of used space on device
20	bits 32 - 63 of used space on device

From RISC OS 3.6 onwards, the Free module calls this reason code to find the free space, rather than calling reason code 2. If R0 is non-zero on exit (ie unaltered), or if an error is generated, the Free module then calls reason code 2. Thus when returning an error from this reason code, your free space routine must also return the same error for reason code 2 before the Free module believes it to be an error.

119 Writing a filing system

Introduction and Overview

New FSEntry_Func and ImageEntry_Func reason codes

In RISC OS 3.6 OS_FSControl has three new reason codes (see *FileSwitch* on page 5a-167), each of which duplicates previously available functionality, but allows 64 bit values to be passed or returned instead of 32 bit values. For each of the new OS_FSControl reason codes, a corresponding new reason code has therefore been added to those that may be passed to a filing system's FSEntry_Func entry point and to an image filing system's ImageEntry_Func entry point:

OS_FSControl	FS/ImageEntry_Func	Name
55	35	ReadFreeSpace64
56	36	DefectList64
57	37	AddDefect64

These new reason codes are detailed in the next section.

Although ImageEntry_Func entry points have been defined, there is little point in an image filing system supporting them under RISC OS 3.6. Since an image filing system is restricted in size to the maximum file size of 4 GB, all quantities can be represented in 32 bits, and the old reason codes are therefore adequate. All programs calling the new 64 bit SWIs (and hence the new entry points) should revert to calling the old 32 bit SWIs (and hence the old entry points) if they get an error; so you shouldn't get problems with new software failing to work because you don't provide the new entry points.

Interfaces

FSEntry_Func 35 and ImageEntry_Func 35

Read free space

On entry

R0 = 35
R1 = pointer to pathname of any object on image (FSEntry_Func 35 only)
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 35); or image filing system's handle for image (ImageEntry_Func 35)

On exit

R0 = bits 0 - 31 of free space
R1 = bits 32 - 63 of free space
R2 = biggest object creatable
R3 = bits 0 - 31 of disc size
R4 = bits 32 - 63 of disc size

Details

This entry point is called by FileSwitch to read the free space for the image that holds the object specified by R1 (FSEntry_Func 35), or that is specified by the handle in R6 (ImageEntry_Func 35).

This entry point is not called by RISC OS 3.5 or earlier.

FSEntry_Func 36 and ImageEntry_Func 36

Read defect list

On entry

R0 = 36
R1 = pointer to name of image (FSEntry_Func 36 only)
R2 = pointer to buffer
R5 = length of buffer
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 36); or image filing system's handle for image (ImageEntry_Func 36)

On exit

R0 preserved
R1 = number of defects placed in buffer
R2, R5, R6 preserved

Details

This entry point is called by FileSwitch to request that your filing system fills the given buffer with a defect list giving the byte offsets to the start of any defects in the specified image. Each entry in the list is a pair of words – with the least significant one first – giving the address of the defect as a 64 bit value.

It is an error if the specified image is not the root object in an image (eg it is an error to map out a defect from adfs::HardDisc4.$.fred, but not an error to map it out from adfs::HardDisc4.$).

This entry point is not called by RISC OS 3.5 or earlier.

FSEntry_Func 37 and ImageEntry_Func 37

Add a defect

On entry

R0 = 37
R1 = pointer to name of image (FSEntry_Func 37 only)
R2 = bits 0 - 31 of offset to start of defect
R3 = bits 32 - 63 of offset to start of defect
R6 = pointer to special field if present, otherwise 0 (FSEntry_Func 37); or image filing system's handle for image (ImageEntry_Func 37)

On exit

R0 - R2, R6 preserved

Details

This entry point is called by FileSwitch to request that your filing system maps out the given defect from the specified image.

It is an error if the specified image is not the root object in an image (eg it is an error to map out a defect from adfs::HardDisc4.$.fred, but not an error to map it out from adfs::HardDisc4.$). If the defect cannot be mapped out because it is not free, then you should return an error.

This entry point is not called by RISC OS 3.5 or earlier.

120 Writing a FileCore module

Under RISC OS 3.6 FileCore has been extended to support larger discs than the versions previously supplied with RISC OS. This has been done by using sector addresses in the interface to FileCore modules, rather than the byte addresses previously used.

Obviously you cannot use these larger discs without both the new version of FileCore, and a new-style FileCore module that supports sector addressing. We therefore recommend that you write all new-style FileCore modules to use sector addressing internally. Existing old-style FileCore modules must be rewritten to use sector addressing before you can use them with large discs.

Declaring your module (page 2-597)

The process for registering a FileCore module has been extended so that each of FileCore and the registering FileCore module can determine whether the other supports sector addressing. You must use this process whenever you register a new-style FileCore module, including on module reinitialisation.

Descriptor block (page 2-597)

The first stage in registering a new-style FileCore module is the same as before; your module must call FileCore_Create (page 2-228). Two new flag bits have been defined in the descriptor block it passes:

Bit	Meaning when set
8	FileCore module supports MiscOp 7
9	FileCore module uses sector offsets (ie uses the new FileCore format)

All versions of FileCore that do not support MiscOp 7 or sector addressing simply ignore these flag bits.

Ensuring the new FileCore is present

Immediately you have a registered a new-style FileCore module using FileCore_Create, you must then check it is running under the new FileCore. To do so, you should call FileCore_MiscOp 6 (page 5a-176) using the module's newly issued private word.

If no error is returned, you may then assume you are running under a FileCore that understands sector addressing, and can take full advantage of the new larger disc sizes available. See *Running new-style FileCore modules under the new FileCore* on page 5a-266.

If an error is returned, the FileCore module is running under an old version of FileCore. You must then either:

- Generate an error stating your module cannot run, and deregister the module; or:
- Set an internal flag to force a backwards compatible mode, and run within the limitations of the old FileCore. See *Running new-style FileCore modules under an old FileCore* on page 5a-268.

Running new-style FileCore modules under the new FileCore

DiscOp entry (page 2-602)

All disc addresses FileCore passes to the DiscOp entry point are sector addresses, rather than the byte addresses used by older versions of FileCore. Since your new-style module should be using sector addresses internally, you shouldn't need to translate these.

MiscOp entry (page 2-604)

The only MiscOp entry point which takes a disc address is MiscOp 0 (Mount). FileCore can't know the sector size until after it has mounted the disc, so it can't pass a sector address to MiscOp 0. Thus the parameters passed to your module's MiscOp entry point are unchanged from earlier versions of RISC OS. The disc address of the boot block (&C00) is still passed as a byte address, and it is your module's responsibility to deal with this.

Returning errors (page 2-600)

The meaning of R0 when returning an error has been extended so that sector addresses can be returned. From RISC OS 3.6 onwards, if bits 30 and 31 of R0 are set, then bits 0 - 29 point to a two-word block:

Offset	Meaning
0	bits 0 - 7 are error number, bits 8 - 29 are clear
4	bits 0 - 28 are the sector number of the disc address, bits 29 - 31 are the drive number

Calling FileCore DiscOp SWIs

Your module is responsible for ensuring that any calls it makes to FileCore's DiscOp SWIs use the correct form of addressing:

- FileCore_SectorOp uses sector addressing; your module doesn't need to translate its own internal sector addresses.
- FileCore_DiscOp continues to use byte addressing; your module must translate between its own internal sector addresses and FileCore_DiscOp's byte addresses, both on entry and on exit.

  Since FileCore has to do a similar translation back to sector addresses before calling your module's low-level entry points, calling FileCore_DiscOp is inefficient, and your module should always use FileCore_SectorOp in preference.

Providing a SWI handler

Your module should provide a full SWI interface, including equivalents to **all** relevant FileCore SWI calls – both new and old. Any calls others make to your module's SWI handler will already use the correct form of addressing for the SWI being called, so typically your handler just needs to set R8 to point to your module's FileCore instance private word, and then call the equivalent FileCore SWI. It does not need to perform any address translation.

Running new-style FileCore modules under an old FileCore

Low-level entry points

If you are running your new-style FileCore module under a version of FileCore that does not understand sector addressing, FileCore will call your low-level entry points using the only form of addressing it knows about: byte addressing. See *Module interfaces* on page 2-602.

Calling FileCore SWIs

Your module must ensure that it only uses calls available under the old FileCore it is using. In particular this means that you must not call FileCore_SectorOp, but must instead use FileCore_DiscOp, translating between your module's own internal sector addresses and FileCore_DiscOp's byte addresses, both on entry and on exit.

Providing a SWI handler

Your module should still provide a full SWI interface, including equivalents to **all** relevant FileCore SWI calls – both new and old. Your SWI handler should downgrade any call to an unavailable 64 bit / sector-addressing SWI to instead call the corresponding 32 bit / byte-addressing FileCore SWI, and then fake the return values for the original 64 bit / sector-addressing call. Thus:

- Calls to FileCore_SectorOp (page 5a-178) should be downgraded to use FileCore_DiscOp (page 2-223)
- Calls to FileCore_FreeSpace64 (page 5a-183) should be downgraded to use FileCore_FreeSpace (page 2-231).

Your module's handler must not attempt to validate reason codes passed to its own DiscOp and MiscOp SWIs; you must – as usual – just set R8 to point to your module's FileCore instance private word, and then call the equivalent FileCore SWI. FileCore is responsible for faulting any unavailable reason codes, such as an attempt to call FileCore_MiscOp 6.

121 Econet

Introduction and Overview

The Econet module was removed from RISC OS 3.5, and is now supplied in a ROM on the Econet network card. (This ROM may also contain updated versions of other RISC OS networking modules.) The first issue of the Econet card uses the Econet 5.70 module, which is the version described below.

New SWIs

The Econet module has had new SWIs added to it. These are documented on the following pages.

Machine type numbers

A machine type number has been defined for machines using the Risc PC architecture, and two further types have been allocated to a third party. In the section *Machine type numbers* on page 2-644, the line:

&000F to &FFF9	Reserved

should now read:

&000F	Risc PC architecture
&0010 to &FFF7	Reserved
&FFF8	SJ Research GP server
&FFF9	SJ Research 80386 UNIX

Port numbers

From RISC OS 3.5 onwards, Econet uses its port numbers as follows:

Port	**Allocation**
&00	Reserved
&01 - &0F	Fixed reply ports, for backward compatibility
&10 - &8F	Dynamic ports, allocated by Econet_AllocatePort
&90 - &FE	Fixed ports, allocated by Acorn Computers
&FF	Argument to Econet_CreateReceive for wild reception

The section *Port numbers* on page 2-649 is no longer accurate for RISC OS 3.5 onwards:

- Allocation &54 for Digital Services Tape Store is no longer used.
- The port number &D0 has been reallocated in RISC OS 3.5 as PrinterServerDataReply.
- Further fixed port numbers have been allocated to third parties, but for reasons of confidentiality we do not list them here.

Changes to existing SWIs

Econet_AllocatePort (page 2-688)

The port numbers returned by this call now always lie in the range &01 - &8F, rather than the range &01 - &FE used by RISC OS 3.11 and earlier.

SWI Calls

Econet_InetRxDirect (SWI &4001D)

This call is for internal use only. You must not use it in your own code.

Econet_EnumerateMap (SWI &4001E)

Enumerates subnetwork addresses within an AUN site network

On entry

R0 = flags:
all bits reserved (must be 0)
R4 = enumeration reference (0 to start)

On exit

R0 preserved
R1 = net number
R2 = pointer to net name, or 0
R3 = IP subnetwork address
R4 = next enumeration reference, or –1 if no more

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call enumerates subnetwork addresses within an AUN site network. It returns the AUN net names, net numbers and IP addresses of the subnetworks active within an AUN site network, as derived from the Map file located within an AUN gateway.

If R4 is –1 on exit then all subnetworks have been enumerated, and R1 - R3 are undefined. If R4 is –1 on exit from the first call then the calling application is running over a network containing no AUN gateways.

Under native Econet R4 is always returned as –1.

This call is available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

Econet_EnumerateTransmit (SWI &4001F)

Returns the handles of open TxCBs

On entry

R0 = index (1 to start with first transmit block)

On exit

R0 = handle, or 0 if no more transmit blocks

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the handles of open TxCBs. On entry R0 is the index of the TxCB being asked for (1, 2, 3, etc). If the value of R0 is greater than the number of open TxCBs, then the value returned as the handle will be 0, which is an invalid handle.

You should not make this call from an IRQ or event routine as, although it will not fail, the returned information may be inaccurate.

This call is available from RISC OS 3.5 onwards.

Related SWIs

Econet_StartTransmit (page 2-667), Econet_PollTransmit (page 2-669), Econet_AbandonTransmit (page 2-671)

Related vectors

None

Econet_HardwareAddresses (SWI &40020)

Returns the addresses of the Econet hardware and interrupt control registers

On entry

—

On exit

R0 = address of MC68B54 ADLC
R1 = address of FIQ mask register
R2 = bit mask value to use on the FIQ mask register

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 returns the addresses of the Econet hardware and interrupt control registers. It is provided for the internal working of Econet diagnostic software, and is not intended for any other use. The call returns an error if there is no native Econet.

This call is available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

Econet_NetworkParameters (SWI &40021)

On entry

—

On exit

R0 = Econet clock period in ¼µs (eg 20 for a 5µs period), or 0 if no clock
R1 = Econet clock frequency in kHz (eg 200 for a 200kHz frequency), or 0 if no clock
R2, R3 corrupted

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 returns the Econet clock period and frequency. The call returns an error if there is no native Econet.

This call is available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

122 AUN

Introduction

The AUN software that this chapter describes forms the core component of Acorn's new networking strategy, called *Acorn Universal Networking* (AUN). AUN uses an industry standard method of passing data over a network: a family of protocols called TCP/IP.

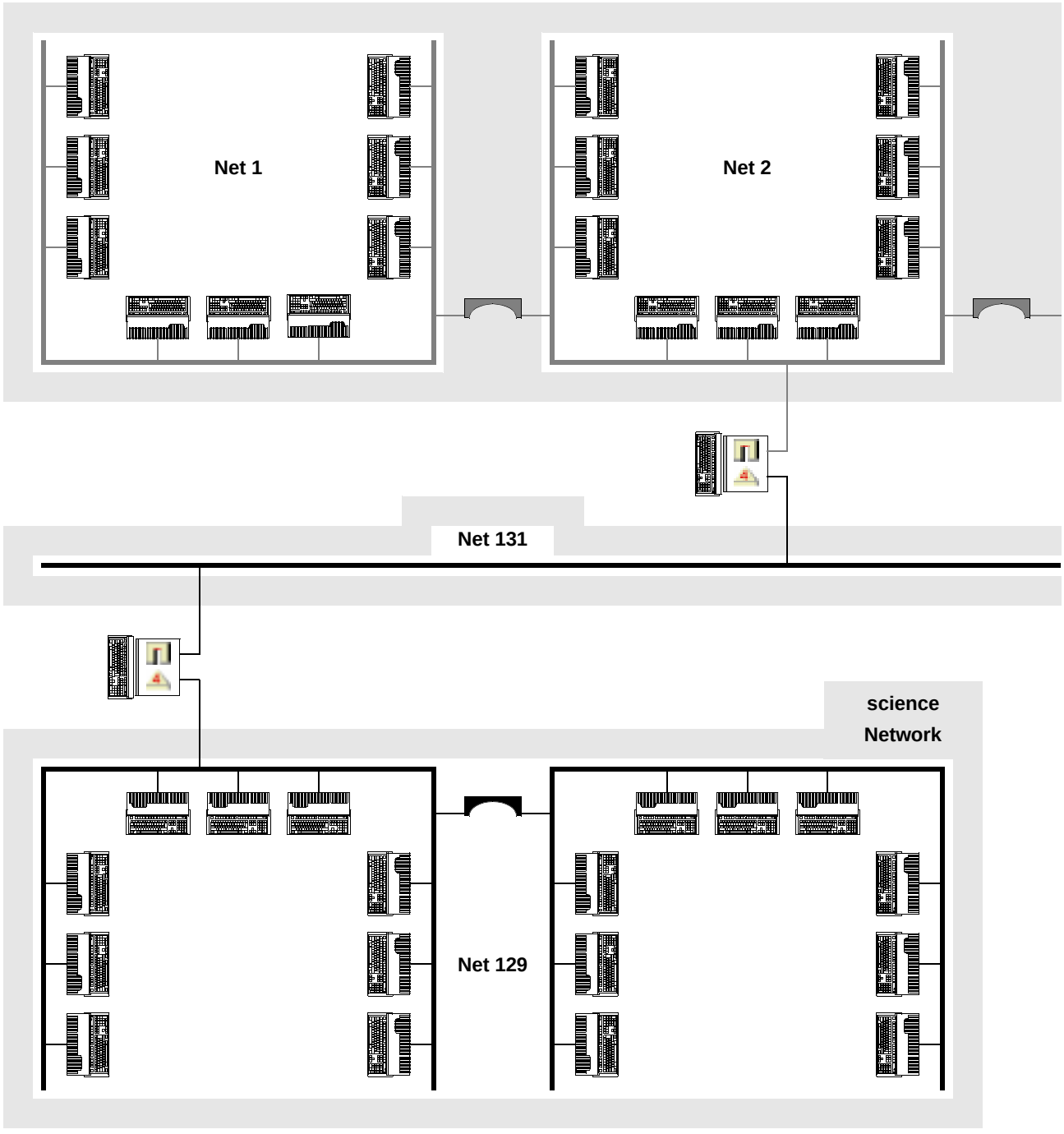
AUN uses the TCP/IP standard in such a way as to retain Econet's existing program interfaces, so your existing network programs should continue to work. Furthermore, AUN's use of the TCP/IP standard supports the concept of Open Systems. Acorn machines – such as Level 4 FileServers – can now co-exist on the same network as other machines that use TCP/IP – such as UNIX workstations and NFS file servers. You can follow this path by using AUN in conjunction with its sister product, the TCP/IP Protocol Suite; this is described in an application note, available from Acorn Customer Services.

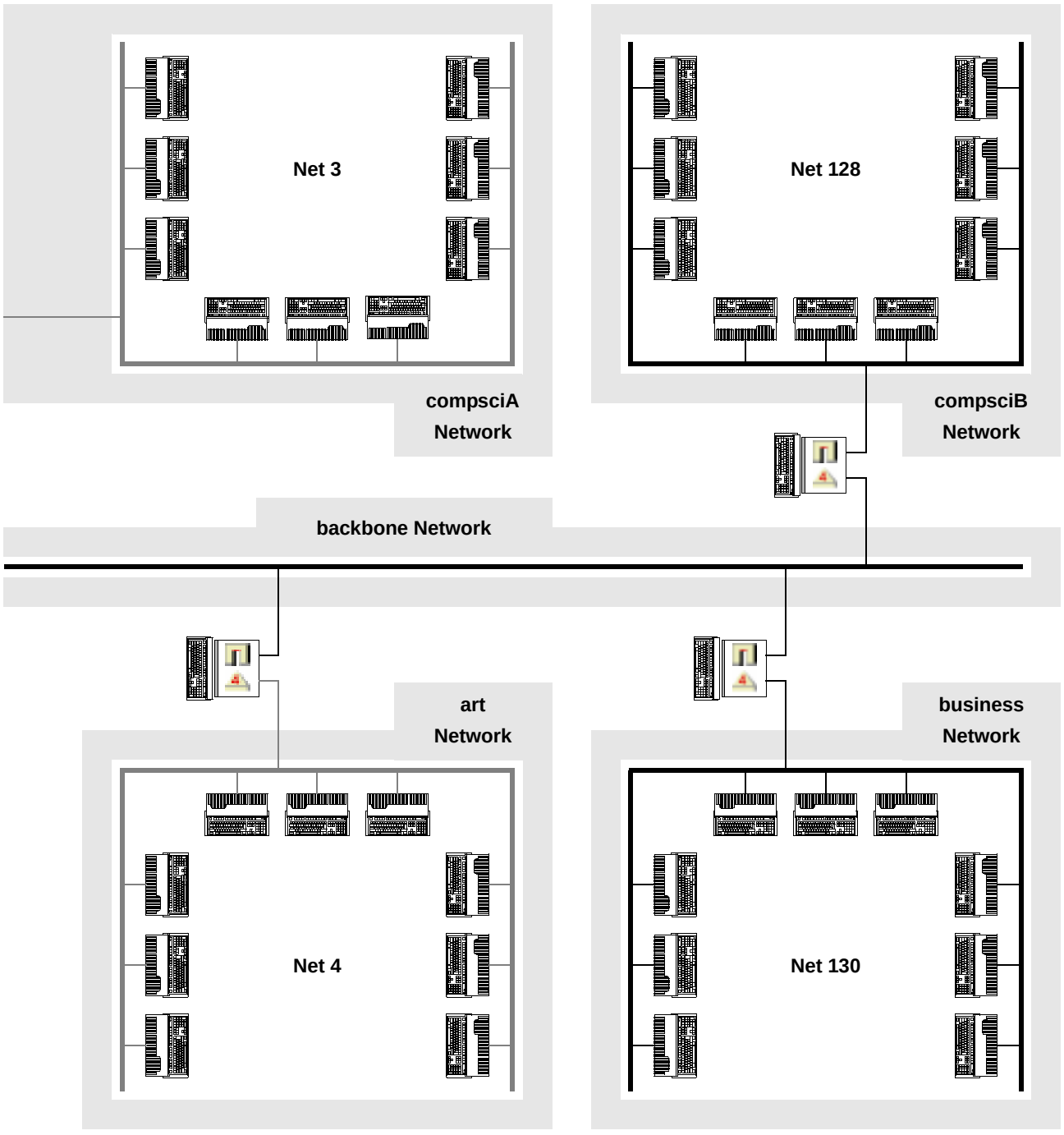
AUN has been designed with an eye to the future, to preserve users' investment as long as possible. In particular, it has been designed so that as new and faster networking technologies become available, developers can easily add support for them by replacing a single hardware-specific module.

For details on using existing Econet networks and AUN networks, refer to the guides supplied with your computer, such as the *RISC OS User Guide*. For details on installing and managing an AUN network, see the *AUN Manager's Guide*.

You should note that networking modules are only loaded if the computer has a network interface fitted.

The rest of this chapter will refer to an example network; this is shown overleaf.


Net 1
Net 2
Net 131
science
Network
Net 129


Net 3
compsciA
Network
Net 128
compsciB
Network
backbone Network
art
Network
Net 4
business
Network
Net 130

Overview

AUN concepts

The basic structure of an AUN site network is one of physically distinct networks, typically associated by location and function with a particular room, department or curriculum area. Adjacent networks are interlinked via gateway stations (described below), which pass messages between the two networks.

Networks

A *network* is a physical network of a single type (e.g. Ethernet, Econet). A network is delimited by any *gateway stations* used to connect it to other networks. For more information on gateway stations, see the section below entitled *Stations*.

Network names

Each network must have a unique name. Network names are not seen or handled by users; they are only used to configure the AUN software for a site.

Nets

A *net* is a part of a network that appears to the user as a single entity.

In both Econet and Ethernet, individual segments of a physical network can be linked together by a *bridge*. However, there is a difference between the two:

- Two bridged Econets remain distinct from each other, and so constitute two distinct nets. Hence in an Econet based network there may be several nets: the initial net, and an extra net for every bridge added.

  For an example see the diagram on page 5a-280. The compsciA network is made up from nets 1, 2 and 3, which are three Econet segments connected by a bridge.

- Two bridged Ethernets appears to users to be a single Ethernet, and so constitute a single net. Hence in an Ethernet based network there will always be one net; in other words, the net and the network are one and the same thing.

  For an example see the same diagram on page 5a-280. The science network and net 129 are identical, and consist of the same two bridged Ethernet segments.

It is important that you grasp the distinction between a net and a network; this chapter will rigorously distinguish between the two.

Net numbers

Each net must have a unique number.

For an Econet the net number must be between 1 and 127.

- If the net is a part of a larger Econet network linked together by bridges, its net number will already be set in the bridge, and the network manager should use the same net number for AUN.
- If the net is not connected to any other Econets (i.e. there aren't any bridges on the net) it will not have a net number assigned to it; under native Econet it will just use the default net number of 0. However, for AUN the network manager must assign it an otherwise unused AUN net number in the permitted range 1 - 127.

For types of net other than Econet (e.g. Ethernet) the net number must be in the range 128 - 252. If such a net is the **only** net on the site (i.e. the whole AUN network consists of a single non-Econet net, such as Ethernet), the network administrator need not set up a net number. It will use net number 128 by default, but – since it is the local net for all stations – users can also refer to it as net 0, in line with Econet convention.

Net numbers 0, 253, 254 and 255 are reserved.

Stations

A station is a computer connected to a net. There are two types of AUN stations.

Client stations

A *client station* has a **single** AUN-configured network interface with which it is connected to a net.

Client stations will form the vast majority of stations in each net, and are typically used as personal workstations.

Gateway stations

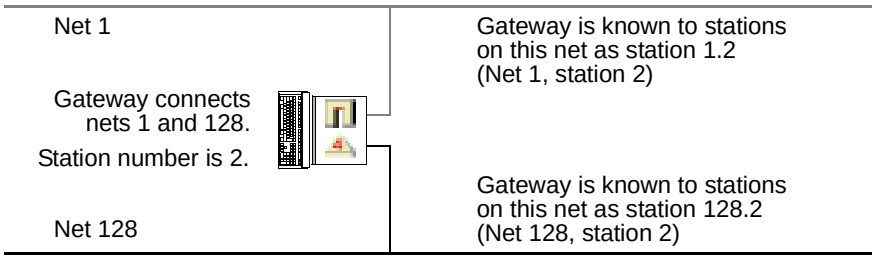
A *gateway station* has **two** AUN-configured network interfaces with which it is connected to a net in each adjacent AUN network. It relays messages between these two networks via the interfaces. The networks may be of different physical types (e.g. Ethernet and Econet). There may only be a single gateway between any two networks.

Station numbers

Each station must have a number, which must be between 2 and 254. Station numbers 0, 1 and 255 are reserved.

A station number must be unique on the net(s) to which the station is connected.

A gateway will have the same station number on both connected nets:



A gateway station’s number must therefore be unused by any other station on either net.

Technical details

Protocols

AUN uses the UDP, IP, ARP, RevARP and RIP protocols from the TCP/IP family:

- The transport protocol is User Datagram Protocol (UDP), enhanced by a proprietary handshake mechanism designed to support the semantics of Econet SWI calls. This is not a straightforward port of the four-way handshake mechanism used by native Econet, but is rather a two-way handshake protocol overlaid with a timeout and retransmission mechanism better suited to the characteristics of IP traffic.

  TCP itself is not used, as it is a stream oriented protocol unsuited to supporting an Econet-like data delivery service.
- The network protocol is Internet Protocol (IP).
- Address Resolution Protocol (ARP) is used to map IP addresses into physical network addresses.
- Reverse Address Resolution Protocol (RevARP) is used by client stations to request their own IP addresses from gateway stations.
- Routing Information Protocol (RIP) is used to pass routing table information between stations.

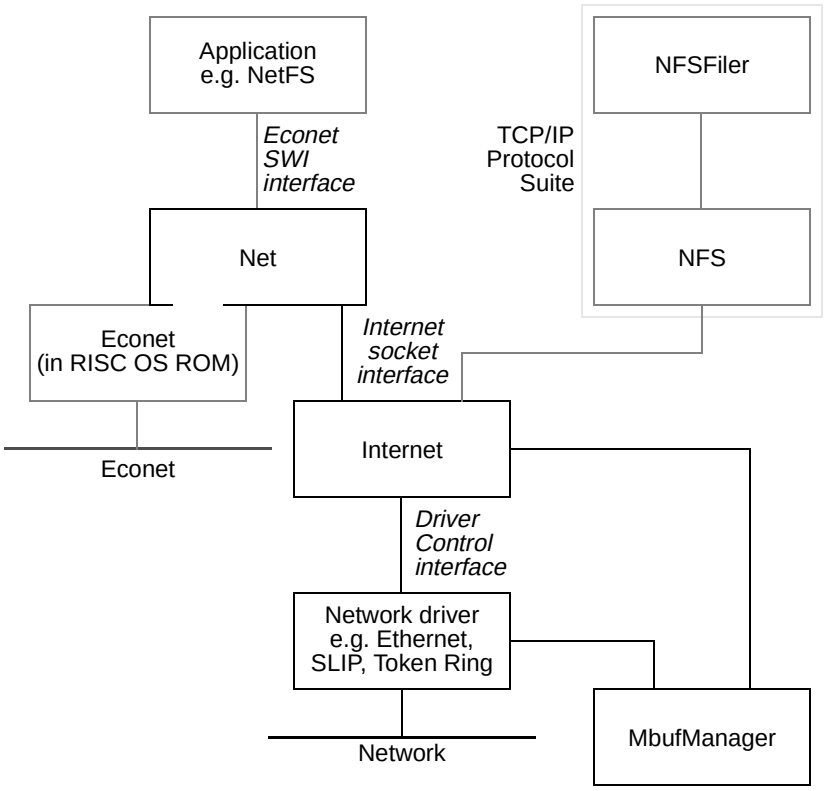
Software

The AUN software consists of several closely related modules:

- The **Net** module implements the two-way acknowledgement handshake, and presents an Econet-like service to applications via Econet SWI calls. It also implements the RIP function.
- The **Internet** module implements UDP, IP, ARP and RevARP protocols, and exports an industry standard (Berkeley socket) interface to other RISC OS software such as the TCP/IP Protocol Suite.
- The **device driver** module provides a *Driver Control Interface* (or *DCI*) that enables the AUN software to communicate with a particular network interface. Each type of network interface needs its own device driver. There are no device drivers supplied in RISC OS 3.5; they are instead normally supplied with network interfaces, either in ROM or on disc.
- The **MbufManager** module provides memory management facilities for version 4 onwards of the DCI, and is used by protocol stacks and device drivers. (It was previously an internal part of the Internet module, and so was not potentially accessible to other protocol modules.)

The software in detail

The following diagram illustrates the relationship between the modules in AUN:



There is a particularly close connection between the Net module and the Econet module. The Net module learns which nets may be accessed via a directly connected Econet, and which nets need to be accessed via IP (ie nets that do not use Econet, or nets using Econet that can only be reached via a gateway). The Net module intercepts SWI calls to Econet from higher-level applications such as NetFS, NetPrint and Broadcast Loader, and – by examining the destination net number – determines whether to route the calls to the Econet module for traffic over native Econet, or to the Internet module for traffic over IP.

If the AUN station does not have an Econet interface fitted then the Econet software module will not be present, and so all traffic will be via the Internet module and IP protocol.

The Internet socket interface – used by the Net module in AUN – remains exposed for parallel use by other applications. Hence other protocols running over IP, such as NFS, can run at the same time as AUN. For more details of the Internet socket interface, see *The Internet module* on page 5a-305.

Since device drivers are not a part of RISC OS itself, we don't document the DCI in this manual. (This also applies to the MbufManager module, which is anyway a conceptual part of the DCI). Both the DCI and the MbufManager module are subject to change as the range of Acorn networking products is expanded and updated. Should you wish to program using the DCI (say to implement a new network interface), you should contact Acorn Customer Services.

Addresses in Econet and AUN

Under native Econet, users and programs uniquely identify each station with two one-byte numbers, thus:

net.station

Under AUN, users and programs use exactly the same scheme, to preserve compatibility with native Econet. However, the underlying Internet protocols used by AUN use four-byte numbers to identify stations. The AUN software therefore needs to translate each two-byte address passed by a user or program into a four-byte IP address. The AUN interpretation of each of the four bytes is:

site.network.net.station

The bottom two bytes (*net.station*) are the same two bytes as are seen by users and programs. The *network* byte is used to provide additional routing information to the underlying IP software only, so that it can route data to the correct destination network. The *site* byte is currently unused and always has a value of one.

Technically speaking, an AUN IP address is a Class A IP address, with a netmask of &FFFF0000.

For example, the AUN interpretation of a command – in the normal IP emphasis – to:

'send data to host 1.3.129.16'

is actually:

'send data to station 129.16… (which is located in network number 3)'

or, more meaningfully:

'send data to station 129.16… (which is located in the science network)'.

The difference between the addressing used by native Econet and the IP address used by AUN is summarised by the table below:

Network	Bytes	Form	Examples
Native Econet address	2	*net.station*	3.2 8.103 129.12
AUN IP address	4	1.*network.net.station*	1.1.3.2 1.4.8.103 1.3.129.12

AUN IP address configuration

How a gateway station finds its full IP address

When a gateway station starts up, it reads its station number from CMOS RAM. (This number is set by the SetStation command supplied with the AUN software.)

To find the site, network and net numbers of both its interfaces, the gateway station looks at its AUN Map file and Configure file.

The Map file

The Map file tells the gateway station the IP address of each net on the site. As an example, let's look at the Map file for the site illustrated on page 5a-280:

```
| Example: Large site network containing 5 dept networks linked via backbone

compsciA        1 2 3                   | old compblock econet
compsciB        128                     | compblock Ethernet
science         129                     | science Ethernet
art             4                       | art room econet
business        130                     | business studies ethernet
backbone        131                     | backbone ethernet
```

The gateway station converts each network name to a network number in the order they're read; the first network has the number 1, the second is number 2, and so on. Adding in the net numbers to the example above, the following full IP addresses apply to the example network. (The site number defaults to 1, and the *station* field is read by each individual station from its configured value in CMOS RAM):

Network name	Network number	Net number	Returned IP address
compsciA	1	1 2 3	1.1.1.*station* 1.1.2.*station* 1.1.3.*station*
compsciB	2	128	1.2.128.*station*
science	3	129	1.3.129.*station*
art	4	4	1.4.4.*station*
business	5	130	1.5.130.*station*
backbone	6	131	1.6.131.*station*

The Configure file

The Configure file tells the gateway station its own position in the site: specifically, which network is connected to which interface. For example:

```
| Example1:
|       network compsciA is Econet;
|       network backbone is Ethernet.

Econet          is compsciA
Slot 0          is backbone
```

This tells the gateway that its Econet interface is connected to the compsciA network, and its Ethernet interface (in slot 0) is connected to the backbone network. What it does not tell the gateway is whether the Econet interface is connected to net 1, 2 or 3. The gateway station resolves this by reading the correct net number (in this case 2) from an Econet bridge on its own net. Thus, if the station number were 7, the two interfaces' IP addresses would be:

1.1.2.7	for the Econet interface
1.6.131.7	for the Ethernet interface

Note that an Ethernet network must always consist of a single net, and so the gateway does not have to resolve the same ambiguities as for Econet.

How a client station finds its full IP address

Like a gateway station, an AUN client station reads its station number from CMOS RAM at start-up time.

However, at this stage it does not know its site, network and net numbers; instead, it finds these out from a gateway station connected to its local network.

To do so the client station broadcasts a RevARP message requesting its IP address. The gateway receives this broadcast on the interface that is connected to the client's network, and returns that interface's IP address, first setting the station number to zero:

site.*network*.*net*.0

Because the gateway station's interface and the client station are on the same network, the returned site and network numbers are therefore the same as those of the client station. The net numbers will also be the same, unless the client station and the gateway station are on different nets within the same network (which can only be the case if they are separated by Econet bridges).

The client station takes the returned address and substitutes its own station number. It also determines if it is connected to a bridged Econet; if so, it replaces the returned net number – which may be incorrect – with the correct net number, read from an Econet bridge on its own net.

Default addresses

If a client station does not get a response to its request for its full IP address, this means that no gateway computer is present and so the local network is isolated. This being the case, then:

- If the station is connected to an Econet it will use native Econet rather than the Internet protocols used by AUN.
- If the station is connected to any other network it adopts a default IP address of 1.0.128.*station*, giving a user address of 128.*station*.

When/if a gateway computer subsequently comes 'on-line' it will immediately send a message to the other stations on the previously isolated network, so they may then complete their address and routing configuration, and get access to all other networks in the AUN system.

Consequently while a network is isolated all its stations may communicate between themselves; stations don't 'hang' awaiting a response from a gateway. You may later start up a gateway station to bring the isolated network into your site's AUN network. However, since this is likely to change 'on the fly' all the addresses of that network's stations, you must take care only to do this when there are no users active on the network.

Application program interface

The application program interface, or API, is the same as the RISC OS 3 (version 3.10) Econet SWI interface, with certain usage qualifications described below. For full details, refer to the *RISC OS 3 Programmer's Reference Manual*.

Existing user applications which access Econet do not require functional modification at the network interface in order to run over an AUN network.

The AUN module intercepts SWI calls to Econet from user software. It treats the calls differently according to how it can access the destination station:

- If the destination station can be accessed directly via Econet, AUN passes the SWI calls to the resident Econet handler. This avoids unnecessary IP protocol overheads for a localised Econet-only transaction.
- Otherwise the destination station must be accessed via IP. AUN maps the SWI calls into calls to the Internet module, having first expanded the two-byte *net.station* destination address into a four-byte *site.network.net.station* IP address.

The maximum amount of data which can be passed in a single transmission SWI via IP is 8192 bytes.

When transmitting to a station via IP, transmission SWI calls will return only the error values Status_NetError and Status_NotListening in the event of failure. Over raw Econet other Econet-specific error values may be returned.

Constraints on the use of Econet SWI calls over AUN

Immediate operations

In general the Immediate mechanism is considered to be Econet specific. The only Immediate operation supported by AUN over IP is Econet_MachinePeek. All other Immediate SWI calls return Status_NotListening, unless the destination station is accessible via a directly connected Econet.

Transmission strategy

An application's choice of values for the Count and Delay parameters it passes to transmission SWIs may make assumptions about the actual physical characteristics of Econet. For example some Econet utility programs set the Count to 0 in Immediate operations, relying on the fact that the return of a scout acknowledge frame in response to a valid scout frame will always be effectively instantaneous. However, over an AUN IP network this assumption is invalid; the functional equivalent of the scout acknowledge may arrive 'sometime', or even 'never'.

Consequently AUN uses a retransmission strategy more suitable to the nature of IP traffic, whilst retaining the existing retransmission strategy for transmissions to a directly connected Econet. The retransmission strategy for AUN over IP is as follows:

For ordinary data, AUN employs a two-way handshake. A receiving station will return a positive acknowledgement if it has successfully received a data frame into an open receive block, or else a reject message if there is currently no open receive block, or some other detectable reception error has occurred.

If Count > 1

The maximum elapsed timeout period in seconds (T) requested by the application is computed as:

T = (Count × Delay) / 100.

On receipt of reject messages, the sender will retransmit the data frame 10 times after 1 centisecond timeouts, then:

If T < 5

T × 10 retransmissions will occur, each after 10 centisecond timeouts;

Else

If the destination station is not on the same network as the sender

exactly 50 retransmissions will occur, each after (T × 100) / 50 centisecond timeouts;

Else

If the retry delay < 25 centiseconds

exactly 50 retransmissions will occur;

Else

(T × 4) retransmissions will occur, each after a 25 centisecond timeout.

(This provides some optimisation for simultaneous loading of software from a local file server, whilst protecting against excessive overload at gateway stations caused by rapid retransmission.)

If no response is received at all then:

If T < 5

1 retransmission will occur, after a 5 second timeout;

Else

T / 5 retransmissions will occur, each after 5 second timeouts.

Else

The sender will transmit exactly once. The transmission status will not change until a positive acknowledgement or a reject message has been received, or a 5 second timeout has elapsed.

For an Immediate operation (i.e. Econet_MachinePeek), a SWI call with Count = 0 or Count = 1 always results in a Status_NotListening return; no actual network transmission is made. In other cases the sender transmits an Immediate message exactly once, changing transmission status only when a response has been received or a 5 second timeout has elapsed.

Bridge protocol

Use of the Econet Bridge protocol by a RISC OS net utility program to identify valid net numbers does not work over non-Econet networks within an AUN system, as no actual Econet bridges are present to respond. However, cycling through the range of net numbers in a sequence of calls to Econet_ReadTransportType can provide this information without involving any network transactions; the call returns R2 = 0 if the given net number is not currently accessible from the local station.

Note that this constraint does not affect use of the Bridge protocol onto a directly connected Econet system.

Meaning of net 0

In AUN, a station may be connected to both an Econet and an Ethernet at the same time. This means that the assumption that Net 0 means the local network is no longer safe, as the AUN software could not, in this case, distinguish the two connected networks with certainty. Hence applications running over AUN should strive to supply an actual net number with every transmission SWI call.

You should note that the actual net number of a connected Econet may in fact be 0, if there are no bridges present; however the net number of an Ethernet in a correctly configured AUN network can never be 0, so no clash will occur. If a net number of 0 is supplied to a transmission SWI, AUN maps it to the net number of a directly connected net, with Econet taking priority over Ethernet if both are connected.

Local broadcasts

If a station is connected to both Econet and Ethernet, transmit SWI requests for a local broadcast – as issued by Broadcast Loader– are directed to the Econet only.

Data delivery

As with Econet, AUN over IP cannot guarantee that a message apparently correctly received and acknowledged by a receiving station will not be retransmitted if the acknowledgement is lost in transit. Applications using AUN should therefore ensure that they can detect whether a transmission has been repeated. This is usually done by adding a sequence number or bit to transmissions.

* Commands

*Configure BootNet

Sets the configured state for whether or not the AUN software is loaded

Syntax

```
*Configure BootNet On|Off
```

Use

*Configure BootNet sets the configured state for whether or not the AUN software is to be loaded from RISC OS 3.5. Drivers are always loaded from a network interface, irrespective of this configured setting.

You should configure this value to ‘On’ if the station is to be a client station using an AUN-configured network, and to ‘Off’ otherwise (i.e. if the station is to be a gateway station, or to be connected to a TCP/IP-configured network).

The default state is ‘Off’.

Example

```
*Configure BootNet On
```

Related commands

None

**Device*Info

Displays driver module internal statistics

Syntax

```
*EBInfo
```

Use

A **Device*Info command displays detailed information about driver module activity. Note that **this command is supplied by the driver that comes with a network interface**, rather than by RISC OS. Each of the standard Acorn drivers provides such a command:

Command	**driver for:**
*EcInfo	Acorn Econet
*EbInfo	Acorn Ethernet

We expect third party drivers to provide a corresponding command; you should see the documentation supplied for the command name.

It is presented mainly as an aid to trouble-shooting, should you require it.

Example

```
*EBInfo
EtherB interface statistics

eb0: 80C04 Network slot, enabled, hardware address 00:00:A4:10:17:00

packets received = 27735              packets transmitted = 2391
bytes received = 2040394              bytes transmitted = 392460
receive interrupts = 27279            transmit interrupts = 2390

Standard clients:

Frame = &0800, ErrLvl=00, AddrLvl=01, FrmLvl=00
Frame = &0806, ErrLvl=00, AddrLvl=01, FrmLvl=00
Frame = &0835, ErrLvl=00, AddrLvl=01, FrmLvl=00



Log:EtherB messages can appear here
```

Related commands

None

*NetMap

Displays the current AUN map table

Syntax

`*NetMap [net_number]`

Use

*NetMap displays the current AUN map table either for the specified net, or for all nets if no parameter is specified. The map table shows the net number of each net, its name, and its Internet address.

Each station obtains the information held in the map table from a gateway's Map file. Since this file is identical for all gateways on a correctly set up network, the output from this command is the same for all stations, and only varies when the network's layout is altered.

Examples

```
*NetMap 129
129     science         1.3.129.x
```

```
*NetMap
1       compsciA        1.1.1.x
2       compsciA        1.1.2.x
3       compsciA        1.1.3.x
128     compsciB        1.2.128.x
129     science         1.3.129.x
4       art             1.4.4.x
130     business        1.5.130.x
131     backbone        1.6.131.x
```

Related commands

*Networks

*NetProbe

Reports if a remote station is accessible and active

Syntax

`*NetProbe` *`net_number.station_number`*

Parameters

`net_number`	remote station's net number
`station_number`	remote station's station number

Use

*NetProbe reports if a remote station is accessible and active, and hence can be reached from the local station and network. This command does so by sending a control message to the specified station and awaiting a reply.

Examples

```
*NetProbe 128.135
Station present
```

```
*NetProbe 128.201
Station not present
```

Related commands

None

*NetStat

Displays the current status of any network interface(s) configured for AUN

Syntax

```
*NetStat [-a]
```

Parameters

`-a` give all information, rather than simplified version

Use

NetStat displays the current status of any network interface(s) configured for AUN. The optional parameter `-a` gives extra information, including traffic counters and full IP addresses. Known network numbers which are marked with an asterisk (‘’) represent nets in a directly connected Econet network.

Example

```
*NetStat -a
Native Econet     0.5                   information for native Econet

Interface         Econet?               information for first AUN interface
AUN Station       4.5
Full address      1.4.4.5

Interface         EtherB                information for second AUN interface
AUN Station       131.5
Full address      1.6.131.5

Known nets        1     2     3    *4     128   129   130
                  131
                                        information below only given if optional parameter a
supplied
TX stats          Data=0, Immediate=2, Imm_Reply=0, Retry=0
                  Error=20, Data_Ack=5, Data_Rej=0, Broadcast=10
                  (local=0, global=5)

RX stats          Data=5, Immediate=0, Broadcast=0, Discard=0
                  Retry=0, Error=0, Data_Ack=0, Data_Rej=0
                  Imm_Reply=2, Reply_Rej=0

Module status     0140
```

Related commands

None

*NetTraceOff

Turns off a gateway's tracing of routing protocol messages

Syntax

```
*NetTraceOff
```

Use

*NetTraceOff turns off a gateway's generation of trace information about its transmission and reception of routing protocol messages. For more details, see the description of the *NetTraceOn command.

This command is provided by the gateway variant of the AUN module, and is hence only available on gateway stations. It is anyway irrelevant to client stations.

Example

```
*NetTraceOff
```

Related commands

*NetTraceOn

*NetTraceOn

Turns on a gateway's tracing of routing protocol messages

Syntax

```
*NetTraceOn [filename]
```

Parameters

filename	name of file to which to direct output

Use

*NetTraceOn turns on a gateway's generation of trace information about its transmission and reception of routing protocol messages. This information is stored in the given file, or – if none is specified – in the file !Gateway.Trace. You can load the trace file into a text editor such as Edit in the usual way.

To view the default file you will need to open the Gateway application directory; hold down the *Shift* key while you double-click on its icon.

This command is provided by the gateway variant of the AUN module, and is hence only available on gateway stations. It is anyway irrelevant to client stations.

Example

```
*NetTraceOn
```

Example output

```
Fri Mar 27 16:26:06:  ==> 131.123
    compsciB        local
    backbone        local
Fri Mar 27 16:26:17:  ==> 131.5
    compsciB        local
    backbone        local
Fri Mar 27 16:27:31:  ==> 131.150
    compsciB        local
    art             gateway=1
    backbone        local
```

Related commands

*NetTraceOff

*Networks

Displays the current AUN routing table

Syntax

```
*Networks
```

Use

*Networks displays the current AUN routing table. This shows the names of any local networks (i.e. those to which the station is directly connected). It also shows the names of those remote networks that the station knows how to reach, and the gateway that it will use to do so.

The AUN routing table alters as gateways start up and shut down, and so the information returned by this command varies as the state of the network alters.

Examples

```
*Networks                                   a client on the 'backbone' net
art             gateway=131.5               connected to the 'art' net by
backbone        local                       gateway 131.5

*Networks                                   a gateway between the 'art'
art             local                       net and the 'backbone' net
backbone        local                       (i.e. station 131.5 above)
```

Related commands

*NetMap

*SetStation

Sets a station's number

Syntax

`*SetStation [`*`station_number`*`]`

Parameters

`station_number` a station number in the range 2 - 254

Use

*SetStation sets a station's number, storing it in CMOS RAM so it is not lost when the computer is switched off. If no number is specified then one is prompted for. If the new station number given is invalid, then the current station number is preserved.

This command is not a part of the standard AUN software, to prevent users from altering station numbers. It is instead supplied as a separate program on the Support disc of the AUN/Level 4 FileServer distribution, in the ArthurLib directory. You can run this program from the desktop by double-clicking on its icon; a window shows the prompt for the station number.

The number is stored in the same location as is used by Econet to store station numbers. If the station is connected to both an AUN network and a native Econet, it will accordingly use the same station number for both types of network. Altering the station number for one network will alter it for the other.

You can find out a station's current station number by typing at a command line:

`*Help Station` *if Econet is fitted*

or:

`*NetStat` *if AUN is installed*

Examples

```
*SetStation 20
```

```
*SetStation
New station number: 20
```

Related commands

*Help Station

123 The Internet module

Introduction

This chapter gives you the guidance and reference material you need to use the socket level programming interfaces provided by the Internet module. We strongly recommend that you only do so once you have a good understanding of Internet protocols and the use of sockets. You should also note that our support services would prefer not to support the Internet module at a tutorial level, since this does not make the most effective use of their resources.

The Internet module

The Internet module has been derived from the Berkeley networking software that was incorporated into the 4.3 BSD 'Reno' release of UNIX (also known as 'net-1'), and into subsequent variations – including Acorn RISC iX. Consequently, the concepts and (to a large extent) the specifics of the programming interface to the Internet module are identical to those provided under BSD UNIX. Most of the differences between the two are caused by differences between the programming environments provided by RISC OS and by UNIX: for example the mechanisms for asynchronous event notification, the assumptions about task scheduling conventions, and so on.

The version of the Internet module in the RISC OS 3.5 ROM is only a partial implementation of the Internet stack, supporting only those protocols needed by then-existent Acorn products. It uses version 2 of the DCI (*Driver Control Interface*). The Internet module in the RISC OS 3.6 ROM (and later) uses DCI 4, and provides a full implementation of the protocol stack. If you wish to program using the Internet socket interface, you should use the full version of the module; see *Getting the libraries and full Internet module* on page 5a-306.

The Internet C libraries

Acorn has C libraries available to help you program the Internet module, which provide the same calls as are used in BSD Unix networking software. Although the Internet module provides a SWI interface, we strongly recommend that you use the libraries, as they provide many extra facilities. They will make it easier to program, especially when porting software; and will enable you to get help from a wealth of supporting books and materials.

Getting the libraries and full Internet module

The libraries – Inetlib, Socklib and Unixlib – are available from Acorn's FTP site (ftp.acorn.co.uk), or on request from Acorn. There are two versions of each library:

- The filenames ending in `zm` are versions intended for use with modules. They are compiled using the *zps1*, *ff* and *zM* switches in the C compiler, so there is no stack limit checking, function names are not embedded, and they are suitable for linking into relocatable modules.
- The other versions are versions intended for use with standard applications. They are compiled using the *zps0* and *fn* switches in the C compiler (but not the *zM* switch), so there is stack limit checking, function names are embedded, and they are not suitable for linking into relocatable modules.

The Internet application is also available from Acorn's FTP site. This includes the current version of the Internet module, which provides a full implementation of the socket interface.

Contents of this chapter

This chapter describes the library calls we recommend you use, rather than describing the more limited range of SWIs. Its organisation is therefore a little different from other chapters in this manual:

- *Introductory tutorial* on page 5a-308 gives an introductory tutorial to programming with the Internet module using the libraries.
- *Advanced tutorial* on page 5a-324 contains a more advanced tutorial.
- *Protocols* on page 5a-361 describes the protocols used by the Internet module.
- *Library calls* on page 5a-367 details the calls in the Socklib library, the Inetlib library, and the Unixlib library. The section starts with an index of the calls.
- *Service calls* on page 5a-461 describes the service calls used by the Internet module and network device drivers.
- *SWI calls* on page 5a-468 describes how to call the Internet module's Socket_… SWI calls; it refers to the earlier documentation.
- ** Commands* on page 5a-470 describes the * commands provided by the Internet module.

About the tutorial sections

The tutorial sections are derived from sections 7 and 8 of the 4.3 BSD *Unix Programmer's Manual Supplementary Documents 1* (or *PS1*). By comparing the two, experienced Internet programmers will be able to see the changes that have been necessary to port the software to RISC OS.

You should also note that the examples in the tutorials assume a pre-emptive multitasking environment such as UNIX, where even if a call does not return for an indefinite period, other programs continue to run. This is not the case for RISC OS. **The example programs do not necessarily multitask correctly under RISC OS.** Before adapting any of the example code for use in RISC OS, you should be aware of which calls might not return promptly, and why; and you should read *Multitasking* on page 5a-357 to find out how to avoid any problems with such calls.

About the protocol and library call sections

We've deliberately kept the documentation of protocols and library calls as similar as possible to normal 4.3 BSD UNIX documentation, so you can easily see what changes we've had to make to cater for RISC OS. (You'll find the equivalent BSD manual pages in sections 2, 3 and 4 of a 4.3 BSD UNIX online manual.) Note that some section headings have been changed for consistency. The function prototypes have also been made consistent in style. Each prototype includes those header files needed to call the functions; the functions' *Description* may mention other useful header files, such as constants that may be passed to/from functions.

Finding out more…

As well as the tutorials in this chapter, you may also find the following book helpful:

- UNIX Network Programming / W. Richard Stevens. – Englewood Cliffs, NJ, USA: Prentice Hall, 1990.

Introductory tutorial

Introduction

RISC OS offers several choices for interprocess communication. To aid the programmer in developing applications which are comprised of cooperating programs, the different choices are discussed and a series of example programs are presented. These programs demonstrate in a simple way the use of sockets and the use of datagram and stream communication. The intent of this tutorial is to present a few simple example programs, not to describe the networking system in full.

Overview

At the core of interprocess communication are *sockets*, from which one reads, and to which one writes. The use of a socket has three phases: its creation, its use for reading and writing, and its destruction. One can write to a socket without full assurance of delivery, since one can check later to catch occasional failures. Messages between sockets can be kept as discrete units, or merged into a stream. One can ask to read, but insist on not waiting if nothing is immediately available.

This tutorial presents simple examples that illustrate some of the ways of doing interprocess communication in RISC OS. We presume you are familiar with the C programming language, but not necessarily with system calls or with interprocess communication. The tutorial reviews the types of communication that are supported by RISC OS. A series of examples are presented that illustrate programs communicating with each other; they show different ways of establishing channels of communication. Finally, the calls that actually transfer data are reviewed. To clearly present how communication can take place, the example programs have been cleared of anything that might be construed as useful work.

Domains and protocols

If we want to communicate between two independent programs, we would like to have them separately create sockets, and then have messages sent between the sockets. This is often the case when providing or using a service in the system. This is also the case when the communicating programs are on separate machines. In RISC OS one can create individual sockets, give them names and send messages between them.

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses for use. The space from which an address is drawn is referred to as a *domain*. RISC OS supports a single domain for sockets, that will be used in the examples here. This is the Internet domain (or AF_INET, for Address

Format InterNET). The Internet domain is an implementation of the DARPA Internet standard protocols IP/TCP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow communication between machines.

Communication follows some particular ‘style.’ Currently, communication is either through a *stream* or by *datagram.* Stream communication implies several things. Communication takes place across a connection between two sockets. The communication is reliable, error-free, and no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to *socketwrite()* or only part of the data from a single call, if there is not enough room for the entire message, or if not all the data from a large message has been transferred. The protocol implementing such a style will retransmit messages received with errors. It will also return error messages if one tries to send a message after the connection has been broken. Datagram communication does not use connections. Each message is addressed individually. If the address is correct, it will generally be received, although this is not guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read, that is, message boundaries are preserved.

The difference in performance between the two styles of communication is generally less important than the difference in semantics. The performance gain that one might find in using datagrams must be weighed against the increased complexity of the program, which must now concern itself with lost or out of order messages. If lost messages may simply be ignored, the quantity of traffic may be a consideration. The expense of setting up a connection is best justified by frequent use of the connection. Since the performance of a protocol changes as it is tuned for different situations, it is best to seek the most up-to-date information when making choices for a program in which performance is crucial.

A protocol is a set of rules, data formats and conventions that regulate the transfer of data between participants in the communication. In general, there is one protocol for each socket type (stream, datagram, etc.) within each domain. The code that implements a protocol keeps track of the names that are bound to sockets, sets up connections and transfers data between sockets, perhaps sending the data across a network. This code also keeps track of the names that are bound to sockets. It is possible for several protocols, differing only in low level details, to implement the same style of communication within a particular domain. Although it is possible to select which protocol should be used, for nearly all uses it is sufficient to request the default protocol. This has been done in all of the example programs.

One specifies the domain, style and protocol of a socket when it is created. For example, in *Figure 123.1* on page 5a-311 the call to *socket()* causes the creation of a datagram socket with the default protocol in the Internet domain.

Closing sockets

It is particularly important that you ensure your applications close all sockets before quitting, say in an *atexit()* routine. This is only shown in the first example program (*Figure 123.1* on page 5a-311); other examples omit this for reasons of space and clarity.

If an application terminates under RISC OS without closing an open socket, then that socket will remain open indefinitely. This needlessly consumes resources; and it leaves fewer sockets available for other programs to use, since socket descriptors are kept in a single fixed-size table.

Datagrams in the Internet domain

Let us now look at two programs that create sockets separately. The programs in *Figure 123.1* on page 5a-311 and *Figure 123.2* on page 5a-313 use datagram communication rather than a stream. The structure used to name Internet domain sockets is defined in the file "*netinet/in.h*". The definition has also been included in the example for clarity.

Each program creates a socket with a call to *socket()*. These sockets are in the Internet domain. Once a name has been created it is attached to a socket by the system call *bind()*. The routine in *Figure 123.2* uses its socket only for sending messages. It does not create a name for the socket because no other program has to refer to it.

Internet addresses specify a host address (a 32-bit number) and a delivery slot, or port, on that machine. These ports are managed by the system routines that implement a particular protocol. When a message must be sent between machines it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, implicitly, an Internet address is a triple including a protocol as well as the port and machine address. An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the tuple *<protocol, local machine address, local port, remote machine address, remote port>*. An association may be transient when using datagram sockets; the association actually exists during a *send* operation.

The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard value INADDR_ANY. The wildcard value is used in the program in *Figure 123.1*. If a machine has several network addresses, it is likely that messages sent to any of the addresses should be deliverable to a socket. This will be the case if the wildcard value has been chosen. Note that even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address. One can be willing to receive from ‘anywhere’, but one cannot send a message

```
#include <stdio.h>
#include "sys/types.h"
#include "sys/socket.h"
#include "netinet/in.h"

/*
 * In the included file "netinet/in.h" a sockaddr_in is defined as follows:
 * struct sockaddr_in {
 *      short   sin_family;
 *      u_short sin_port;
 *      struct in_addr sin_addr;
 *      char    sin_zero[8];
 * };
 *
 * This program creates a datagram socket, binds a name to it, then reads
 * from the socket.
 */

char buf[1024]; /* global rather than auto, so doesn't go on SVC stack */
int sock = -1;  /* mark socket as initially closed */

finalise()      /* exit handler to close socket, registered with atexit */
{
        if (sock != -1) {               /* if socket not already closed */
                socketclose(sock);      /* close it */
                sock = -1;              /* and mark it as closed */
        }
}

main()
{
        int length;
        struct sockaddr_in name;

        /* Register finalisation code to close socket at exit */
        if (atexit(finalise) != 0) {
                fprintf(stderr, "Unable to register exit handler\n");
                exit (1);
        }

        /* Create socket from which to read. */
        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
                xperror("opening datagram socket");
                exit(1);
        }
```

Figure 123.1 Reading Internet domain datagrams

```
        /* Create name with wildcards. */
        name.sin_family = AF_INET;
        name.sin_addr.s_addr = INADDR_ANY;
        name.sin_port = 0;
        if (bind(sock, &name, sizeof(name))) {
                xperror("binding datagram socket");
                exit(1);
        }

        /* Find assigned port value and print it out. */
        length = sizeof(name);
        if (getsockname(sock, &name, &length)) {
                xperror("getting socket name");
                exit(1);
        }
        printf("Socket has port #%d\n", ntohs(name.sin_port));

        /* Read from the socket */
        if (socketread(sock, buf, 1024) < 0)
                xperror("receiving datagram packet");
        printf("-->%s\n", buf);

        /* Close the socket */
        socketclose(sock);
        sock = -1;                      /* mark it as closed */
}
```

Figure 123.1 Reading Internet domain datagrams (continued)

‘anywhere’. The program in *Figure 123.2* is given the destination host name as a command line argument. To determine a network address to which it can send the message, it looks up the host address by the call to *gethostbyname()*. The returned structure includes the host’s network address, which is copied into the structure specifying the destination of the message.

The port number can be thought of as the number of a mailbox, into which the protocol places one’s messages. Certain daemons, offering certain advertised services, have reserved or ‘well-known’ port numbers. These fall in the range from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system will assign an unused port number when an address is bound to a socket. This may happen when an explicit *bind* call is made with a port number of 0, or when a *connect* or *send* is performed on an unbound socket. Note that port numbers are not automatically reported back to the user. After calling *bind()*, asking for port 0, one may call *getsockname()* to discover what port was actually assigned.

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of the bytes in the address. Because machines differ in the internal representation they ordinarily use to represent integers, printing out the port number as returned by *getsockname()* may result in a misinterpretation. To print out the number, it is necessary to use the routine *ntohs()* (for

```
#include <stdio.h>
#include "sys/types.h"
#include "sys/socket.h"
#include "netinet/in.h"
#include "netdb.h"

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments. The form of the command line is dgramsend hostname
 * portnumber
 */

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_in name;
        struct hostent *hp, *gethostbyname();

        /* Create socket on which to send. */
        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
                xperror("opening datagram socket");
                exit(1);
        }

        /*
         * Construct name, with no wildcards, of the socket to send to.
         * Gethostbyname() returns a structure including the network
         * address of the specified host. The port number is taken from
         * the command line.
         */
        hp = gethostbyname(argv[1]);
        if (hp == 0) {
                fprintf(stderr, "%s: unknown host\n", argv[1]);
                exit(2);
        }
        bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
        name.sin_family = AF_INET;
        name.sin_port = htons(atoi(argv[2]));

        /* Send message. */
        if (sendto(sock, DATA, sizeof(DATA), 0, &name, sizeof(name)) < 0)
                xperror("sending datagram message");

        socketclose(sock);
}
```

Figure 123.2 Sending an Internet domain datagram

network to host: short) to convert the number from the network representation to the host's representation. On some machines, such as 68000-based machines, this is a null operation. On others, such as ARMs and VAXes, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format, called *htons()*; similar routines exist for long integers.

Connections

To send data between stream sockets (having communication style SOCK_STREAM), the sockets must be connected. *Figure 123.3* on page 5a-315 and *Figure 123.5* on page 5a-317 show two programs that create such a connection. The program in *Figure 123.3* is relatively simple. To initiate a connection, this program simply creates a stream socket, then calls *connect()*, specifying the address of the socket to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the program can begin to send messages. Messages will be delivered in order without message boundaries. The connection is destroyed when either socket is closed (or soon thereafter). If a program tries to send messages after the connection is closed, the call will fail, and the errno variable is set to 'EPIPE'.

Forming a connection is asymmetrical; one program, such as the program in *Figure 123.3*, requests a connection with a particular socket, the other program accepts connection requests. Before a connection can be accepted a socket must be created and an address bound to it. This situation is illustrated in the top half of *Figure 123.4* on page 5a-316. Program 2 has created a socket and bound a port number to it. Program 1 has created an unnamed socket. The address bound to Program 2's socket is then made known to Program 1 and, perhaps to several other potential communicants as well. If there are several possible communicants, this one socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this program for this connection. A connection may be destroyed by closing the corresponding socket.

The program in *Figure 123.5* is a rather trivial example of a server. It creates a socket to which it binds a name, which it then advertises. (In this case it prints out the socket number.) The program then calls *listen()* for this socket. Since several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. *Listen()* marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of *listen()*; the maximum length is limited by the system. Once the listen call has been completed, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and, hence, a new socket for the connection is created. The bottom half of *Figure 123.4* shows the result of Program 1 connecting with

```
#include <stdio.h>
#include "sys/types.h"
#include "sys/socket.h"
#include "netinet/in.h"
#include "netdb.h"

#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with the socket
 * given in the command line. One message is sent over the connection and
 * then the socket is closed, ending the connection. The form of the
 * command line is streamwrite hostname portnumber
 */

char buf[1024]; /* global rather than auto, so doesn't go on SVC stack */

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_in server;
        struct hostent *hp, *gethostbyname();

        /* Create socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                xperror("opening stream socket");
                exit(1);
        }

        /* Connect socket using name specified by command line. */
        server.sin_family = AF_INET;
        hp = gethostbyname(argv[1]);
        if (hp == 0) {
                fprintf(stderr, "%s: unknown host\n", argv[1]);
                exit(2);
        }
        bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
        server.sin_port = htons(atoi(argv[2]));

        if (connect(sock, &server, sizeof(server)) < 0) {
                xperror("connecting stream socket");
                exit(1);
        }

        if (socketwrite(sock, DATA, sizeof(DATA)) < 0)
                xperror("writing on stream socket");

        close(sock);
}
```

Figure 123.3 Initiating an Internet domain stream connection

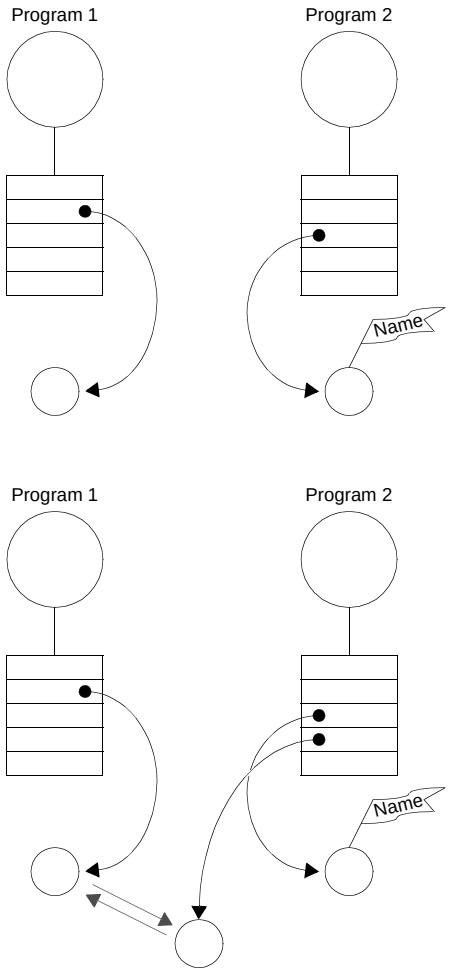


Figure 123.4 Establishing a stream connection

the named socket of Program 2, and Program 2 accepting the connection. After the connection is created, the service, in this case printing out the messages, is performed and the connection socket closed. The *accept()* call will take a pending connection request from the queue if one is available, or block waiting for a request. Messages are read from the connection socket. Reads from an active connection will normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the read call returns immediately. The number of bytes returned will be zero.

```
#include <stdio.h>
#include "sys/types.h"
#include "sys/socket.h"
#include "netinet/in.h"
#include "netdb.h"
#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop. Each
 * time through the loop it accepts a connection and prints out messages
 * from it. When the connection breaks, or a termination message comes
 * through, the program accepts a new connection.
 */

char buf[1024]; /* global rather than auto, so doesn't go on SVC stack */

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        int rval;
        int i;

        /* Create socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                xperror("opening stream socket");
                exit(1);
        }

        /* Name socket using wildcards */
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = 0;
        if (bind(sock, &server, sizeof(server))) {
                xperror("binding stream socket");
                exit(1);
        }

        /* Find out assigned port number and print it out */
        length = sizeof(server);
        if (getsockname(sock, &server, &length)) {
                xperror("getting socket name");
                exit(1);
        }
        printf("Socket has port #%d\n", ntohs(server.sin_port));
```

Figure 123.5 Accepting an Internet domain stream connection

```
        /* Start accepting connections */
        listen(sock, 5);
        do {
                msgsock = accept(sock, 0, 0);
                if (msgsock == -1)
                        xperror("accept");
                else do {
                        bzero(buf, sizeof(buf));
                        if ((rval = socketread(msgsock, buf, 1024)) < 0)
                                xperror("reading stream message");
                        i = 0;
                        if (rval == 0)
                                printf("Ending connection\n");
                        else
                                printf("-->%s\n", buf);
                } while (rval != 0);
                close(msgsock);
        } while (TRUE);
        /*
}
```

Figure 123.5 Accepting an Internet domain stream connection (continued)

The program in *Figure 123.6* on page 5a-319 is a slight variation on the server in *Figure 123.5*. It avoids blocking when there are no pending connection requests by calling *select()* to check for pending requests before calling *accept()*. This strategy is useful when connections may be received on more than one socket, or when data may arrive on other connected sockets before another connection request.

```
#include <stdio.h>
#include "sys/types.h"
#include "sys/socket.h"
#include "sys/time.h"
#include "netinet/in.h"
#include "netdb.h"

#define TRUE 1

/*
 * This program uses select() to check that someone is trying to connect
 * before calling accept().
 */

char buf[1024]; /* global rather than auto, so doesn't go on SVC stack */

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        int rval;
        fd_set ready;
        struct timeval to;

        /* Create socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                xperror("opening stream socket");
                exit(1);
        }

        /* Name socket using wildcards */
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = 0;
        if (bind(sock, &server, sizeof(server))) {
                xperror("binding stream socket");
                exit(1);
        }

        /* Find out assigned port number and print it out */
        length = sizeof(server);
        if (getsockname(sock, &server, &length)) {
                xperror("getting socket name");
                exit(1);
        }
        printf("Socket has port #%d\n", ntohs(server.sin_port));
```

Figure 123.6 Using select() to check for pending connections

```
        /* Start accepting connections */
        listen(sock, 5);
        do {
                FD_ZERO(&ready);
                FD_SET(sock, &ready);
                to.tv_sec = 5;
                if (select(sock + 1, &ready, 0, 0, &to) < 0) {
                        xperror("select");
                        continue;
                }
                if (FD_ISSET(sock, &ready)) {
                        msgsock=accept(sock,(struct sockaddr *)0,(int *)0);
                        if (msgsock == -1)
                                xperror("accept");
                        else do {
                                bzero(buf, sizeof(buf));
                                if ((rval=socketread(msgsock,buf,1024))<0)
                                        xperror("reading stream message");
                                else if (rval == 0)
                                        printf("Ending connection\n");
                                else
                                        printf("-->%s\n", buf);
                        } while (rval > 0);
                        close(msgsock);
                } else
                        printf("Do something else\n");
        } while (TRUE);
}
```

Figure 123.6 Using select() to check for pending connections (continued)

Reads, writes, recvs, etc

Socklib has several system calls for reading and writing information. The simplest calls are *socketread()* and *socketwrite(). Socketwrite()* takes as arguments the index of a descriptor, a pointer to a buffer containing the data and the size of the data. The descriptor indicates a connected socket. 'Connected' can mean either a connected stream socket (as described in *Connections* on page 5a-314) or a datagram socket for which a *connect()* call has provided a default destination (see page 5a-382). *Socketread()* also takes a descriptor that indicates a socket. *Socketwrite()* requires a connected socket since no destination is specified in the parameters of the system call. *Socketread()* can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that require no assumptions about the source of their input or the destination of their output. There are variations on *socketread()* and *socketwrite()* that allow the source and destination of the input and output to use several separate buffers. These are *socketreadv()* and *socketwritev(),* for read and write *vector.*

It is sometimes necessary to send high priority data over a connection that may have unread low priority data at the other end. For example, a user interface program may be interpreting commands and sending them on to another program through a stream connection. The user interface may have filled the stream with as yet unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high priority data wait to be processed after the low priority data, it is possible to send it as *out-of-band* (OOB) data. The notification of pending OOB data results in the generation of an Internet event (see *The Internet event* on page 5a-345). There are a pair of calls similar to *socketread* and *socketwrite* that allow options, including sending and receiving OOB information; these are *send()* and *recv().* These calls also allow *peeking* at data in a stream. That is, they allow a program to read data without removing the data from the stream. One use of this facility is to read ahead in a stream to determine the size of the next item to be read. When not using these options, these calls have the same functions as *socketread()* and *socketwrite().*

To send datagrams, one must be allowed to specify the destination. The call *sendto()* takes a destination address as an argument and is therefore used for sending datagrams. The call *recvfrom()* is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data. If the identity of the sender does not matter, one may use *socketread()* or *recv().*

Finally, there are a pair of calls that allow the sending and receiving of messages from multiple buffers, when the address of the recipient must be specified. These are *sendmsg()* and *recvmsg().*

The various options for reading and writing are shown in *Figure 123.7* on page 5a-322, together with their parameters. The parameters for each system call reflect the differences in function of the different calls. In the examples given in this tutorial, the calls *socketread()* and *socketwrite()* have been used whenever possible.

```
/*
 * The variable "sock" must be the descriptor of a socket.
 */
cc = socketread(sock, buf, nbytes)
int cc, sock; char *buf; int nbytes;

/*
 * An iovec can include several source buffers.
 */
cc = socketreadv(sock, iov, iovcnt)
int cc, sock; struct iovec *iov; int iovcnt;

cc = socketwrite(sock, buf, nbytes)
int cc, sock; char *buf; int nbytes;

cc = socketwritev(sock, iovec, ioveclen)
int cc, sock; struct iovec *iovec; int ioveclen;

/*
 * Flags may include MSG_OOB and MSG_PEEK.
 */
cc = send(sock, msg, len, flags)
int cc, sock; char *msg; int len, flags;

cc = sendto(sock, msg, len, flags, to, tolen)
int cc, sock; char *msg; int len, flags;
struct sockaddr *to; int tolen;

cc = sendmsg(sock, msg, flags)
int cc, sock; struct msghdr msg[]; int flags;

cc = recv(sock, buf, len, flags)
int cc, sock; char *buf; int len, flags;

cc = recvfrom(sock, buf, len, flags, from, fromlen)
int cc, sock; char *buf; int len, flags;
struct sockaddr *from; int *fromlen;

cc = recvmsg(sock, msg, flags)
int cc, sock; struct msghdr msg[]; int flags;
```

Figure 123.7 Varieties of socketread and socketwrite commands

Choices

This introductory tutorial has presented examples of some of the forms of communication supported by RISC OS. These have been presented in an order chosen for ease of presentation. It is useful to review these options emphasizing the factors that make each attractive.

The Internet domain allows communication between machines. This makes the Internet domain a necessary choice for programs running on separate machines.

The choice between datagrams and stream communication is best made by carefully considering the semantic and performance requirements of the application. Streams can be both advantageous and disadvantageous. One disadvantage is that a program is only allowed a limited number of open streams, as there are usually only 96 entries available in the system-wide open descriptor table. This can cause problems if a single server must talk with a large number of clients. Another is that for delivering a short message the stream setup and teardown time can be unnecessarily long. Weighed against this is the reliability built into the streams. This will often be the deciding factor in favour of streams.

What to do next

Many of the examples presented here can serve as models for multiprocess programs and for programs distributed across several machines. In developing a new multiprocess program, it is often easiest to first write the code to create the programs and communication paths. After this code is debugged, the code specific to the application can be added.

Advanced tutorial

Introduction

This section gives you a more advanced tutorial on the communications programming facilities provided by the Internet module. It looks at the overall model for communication, outlines the communications primitives we've provided, and (in particular) looks at how to use these primitives in developing applications.

This tutorial provides a high-level description of the communications facilities and their use. It is complements the descriptions of the library calls later in this chapter by examples of their use. The remainder of this section is organized in parts:

- *Basics* on page 5a-325 introduces the communication-related calls and the basic model of communication.
- *Network library routines* on page 5a-334 describes some of the supporting library routines users may find useful in constructing distributed applications.
- *Client/server model* on page 5a-339 is concerned with the client/server model used in developing applications, and includes examples of the two major types of servers.
- *The Internet event* on page 5a-345 describes the Internet event which is used by a number of important features, such as asynchronous I/O, and out-of-band data.
- *Advanced topics* on page 5a-348 delves into advanced topics which sophisticated users are likely to encounter when using the communications facilities.
- *Multitasking* on page 5a-357 outlines how to ensure that programs using the Internet module multitask correctly under RISC OS. **It is essential that you read this section and follow its recommendations**.

You should be familiar with the C programming language, as all examples are written in C.

Basics

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which you can *bind* a name. Each socket in use has a *type*.

Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of programs communicating through sockets. One such property is the scheme used to name sockets. Sockets normally exchange data only with sockets in the same domain. (It may be possible to cross domain boundaries, but only if some translation process is performed.)

The RISC OS socket subsystem currently only supports a single communication domain: the *Internet domain*, which is used by programs which communicate using the DARPA standard communication protocols.

Socket types

Sockets are typed according to the communication properties visible to a user. Programs are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets currently are available to a user.

- A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. (Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of BSD UNIX pipes.)
- A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a program receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.
- A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in *Selecting specific protocols* on page 5a-350.

Socket creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular *protocol* may also be requested.

- The domain is specified as one of the manifest constants defined in the file *"sys/socket.h"*. The manifest constants are named AF_… as they indicate the ‘address format’ to use in interpreting names; for the Internet domain supported by RISC OS the constant is AF_INET.
- The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW must be specified.
- If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type.

The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets.

To create a stream socket in the Internet domain the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for the Internet domain use the call might be:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The default protocol (used when the *protocol* argument to the *socket* call is 0) should be correct for most every situation. However, it is possible to specify a protocol other than the default; this is covered in *Selecting specific protocols* on page 5a-350.

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

Binding local names

A socket is created without a name. Until a name is bound to a socket, programs have no way to reference it and, consequently, no messages may be received on it. Communicating programs are bound by an *association*. In the Internet domain, an association is composed of local and foreign Internet addresses, and local and foreign port numbers. In most domains, associations must be unique. In the Internet domain there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The *bind* system call allows a program to specify half of an association, <local address, local port>, while the *connect* and *accept* primitives are used to complete a socket's association.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the *connect* and *send* calls will automatically bind an appropriate address if they are used with an unbound socket.

The *bind* system call is used as follows:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the domain). As mentioned, in the Internet domain names contain an Internet address and port number. If one wanted to bind an Internet address, the following code would be used:

```
#include "sys/types.h"
#include "netinet/in.h"
...
struct sockaddr_in sin;
...
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in *Network library routines* on page 5a-334, when the library routines used in name resolution are discussed.

Connection establishment

Connection establishment is usually asymmetric, with one program a *client* and the other a *server.*

- The server, when willing to offer its advertised services, binds a socket to a well-known address associated with the service and then passively 'listens' on its socket.

  It is then possible for an unrelated program to rendezvous with the server.

- The client requests services from the server by initiating a ‘connection’ to the server’s socket.

On the client side the *connect* call is used to initiate a connection. Using the Internet domain, this might appear as:

```
struct sockaddr_in server;
...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

where *server* in the example above contains the Internet address and port number of the server to which the client program wishes to speak.

If the client program’s socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin. Some of the more common errors returned when a connection attempt fails are:

ETIMEDOUT	After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.
ECONNREFUSED	The host refused service for some reason. This is usually due to a server program not being present at the requested name.
ENETDOWN or EHOSTDOWN	These operational errors are returned based on status information delivered to the client host by the underlying communication services.
ENETUNREACH or EHOSTUNREACH	These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down, in which case the system indicates the entire network is unreachable.

For the server to receive a client’s connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server program; this number may be limited by the system. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is currently limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of programs hogging system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server may *accept* a connection:

```
struct sockaddr_in from;
...
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

Accept normally blocks. That is, *accept* will not return until a connection is available or the system call is interrupted – for example by Escape being pressed. Further, there is no way for a program to indicate it will accept connections from only a specific individual, or individuals. It is up to the user program to consider who the connection is from and close down the connection if it does not wish to speak to the program. If the server program wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; they will be considered in *Advanced topics* on page 5a-348.

Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. In this case the *socketread* and *socketwrite* system calls are usable:

```
socketwrite(s, buf, sizeof (buf));
socketread(s, buf, sizeof (buf));
```

In addition to *socketread* and *socketwrite*, the calls *send* and *recv* may be used:

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While *send* and *recv* are virtually identical to *socketread* and *socketwrite*, the extra *flags* argument is important. The flags, defined in *"sys/socket.h"*, may be specified as a non-zero value if one or more of the following is required:

MSG_OOB	send/receive out-of-band data
MSG_PEEK	look at data without reading
MSG_DONTROUTE	send data without routing packets

- Out-of-band data is a notion specific to stream sockets, and one which we will not immediately consider.
- The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management program, and is unlikely to be of interest to the casual user.
- The ability to preview data is, however, of interest. When MSG_PEEK is specified with a *recv* call, any data present is returned to the user, but treated as still ‘unread’. That is, the next *socketread* or *recv* call applied to the socket will return the data previously previewed.

Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *socketclose* to the descriptor:

```
socketclose(s);
```

If data is associated with a socket which promises reliable delivery (eg a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

When a client or server machine crashes, the socket stays open on the machine that hasn’t crashed. Afterwards, under RISC OS, *socketwrite* or *send* calls will result in an event being generated (see *The Internet event* on page 5a-345) and a return error of EPIPE, *socketread* or *recv* calls will return an EOF indication.

Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram

socket provides a symmetric interface to data exchange. While programs are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before. If a particular local address is needed, the *bind* operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent.

To send data, the *sendto* primitive is used:

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

- The *s*, *buf*, *buflen*, and *flags* parameters are used as before.
- The *to* and *tolen* values are used to indicate the address of the intended recipient of the message.

When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return –1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, the *recvfrom* primitive is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

- Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the *connect* call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time; a second connect will change the destination address, and a connect to a null address (family AF_UNSPEC) will disconnect. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end to end connection). *Accept* and *listen* are not used with datagram sockets.

While a datagram socket is connected, errors from recent *send* calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with *getsockopt*, SO_ERROR, may be used to interrogate the error status. A *select* for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. Other of the less important details of datagram sockets are described in *Advanced topics* on page 5a-348.

Input/output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets. This is done using the *select* call:

```
#include "sys/time.h"
#include "sys/types.h"
...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

Select takes as arguments pointers to three sets:

- one for the set of socket descriptors for which the caller wishes to be able to read data on
- one for those descriptors to which data is to be written
- one for which exceptional conditions are pending

  (Out-of-band data is the only exceptional condition currently implemented by the socket. If the user is not interested in certain conditions – ie read, write, or exceptions – the corresponding argument to the *select* should be a null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition FD_SETSIZE. The array must be long enough to hold one bit for each of FD_SETSIZE descriptors.

The macros FD_SET(*fd, &mask*) and FD_CLR(*fd, &mask*) have been provided for adding and removing descriptor *fd* in the set *mask*. The set should be zeroed before use, and the macro FD_ZERO(*&mask*) has been provided to clear the set *mask*.

The parameter *nfds* in the *select* call specifies the range of descriptors (ie one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely.

Select normally returns the number of descriptors selected; if the *select* call returns due to the timeout expiring, then the value 0 is returned. If the *select* terminates because of an error or interruption, a –1 is returned with the error number in *errno*, and with the socket descriptor masks unchanged.

Assuming a successful return, the three sets will indicate which descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a socket descriptor in a select mask may be tested with the *FD_ISSET(fd, &mask)* macro, which returns a non-zero value if *fd* is a member of the set *mask*, and 0 if it is not.

To determine if there are connections waiting on a socket to be used with an *accept* call, *select* can be used, followed by a *FD_ISSET(fd, &mask)* macro to check for read readiness on the appropriate socket. If *FD_ISSET* returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

As an example, to read data from two sockets, *s1* and *s2* as it is available from each and with a one-second timeout, the following code might be used:

```
#include "sys/time.h"
#include "sys/types.h"
...
fd_set read_template;
struct timeval wait;
...
for (;;) {
        wait.tv_sec = 1;        /* one second */
        wait.tv_usec = 0;

        FD_ZERO(&read_template);

        FD_SET(s1, &read_template);
        FD_SET(s2, &read_template);

        nb = select(FD_SETSIZE,&read_template,(fd_set *)0,(fd_set *)0,&wait);
        if (nb <= 0) {
                An error occurred during the select, or the select timed out.
        }
        if (FD_ISSET(s1, &read_template)) {
                Socket #1 is ready to be read from.
        }
        if (FD_ISSET(s2, &read_template)) {
                Socket #2 is ready to be read from.
        }
}
```

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the Internet events described in *The Internet event* on page 5a-345.

Network library routines

The discussion in the previous part of this tutorial indicated the possible need to locate and construct network addresses when using the communication facilities in a distributed environment. To aid in this task a number of routines have been provided in the Inetlib library. In this section we will consider the routines provided to manipulate network addresses.

Locating a service on a remote host requires many levels of mapping before client and server may communicate:

- A service is assigned a name which is intended for human consumption; eg ‘the *login server* on host monet’.
- This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption.
- Finally, the address must then used in locating a physical *location* and *route* to the service.

The specifics of these three mappings are likely to vary between network architectures. For instance, it is desirable for a network to not require hosts to be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for mapping:

- host names to network addresses
- network names to network numbers
- protocol names to protocol numbers
- service names to port numbers and the appropriate protocol to use in communicating with the server program.

The file *"netdb.h"* must be included when using any of these routines.

Host names

An Internet host name to address mapping is represented by the *hostent* structure:

```
struct hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type (eg AF_INET) */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses, null terminated */
};

#define h_addr  h_addr_list[0]  /* first address, network byte order */
```

The routine *gethostbyname* takes an Internet host name and returns a *hostent* structure, while the routine *gethostbyaddr* maps Internet host addresses into a *hostent* structure.

The official name of the host and its public aliases are returned by these routines, along with the address type (family) and a null terminated list of variable length addresses. This list of addresses is required because it is possible for a host to have many addresses, all having the same name. The *h_addr* definition is provided for backward compatibility, and is defined to be the first address in the list of addresses in the *hostent* structure.

The database for these calls is provided by the file *InetDBase:hosts*. When using *gethostbyname*, only one address will be returned, but all listed aliases will be included.

Network names

As for host names, routines for mapping network names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits -- probably a poor one.
*/
struct  netent {
        char    *n_name;        /* official name of net */
        char    **n_aliases;    /* alias list */
        int     n_addrtype;     /* net address type */
        int     n_net;          /* network number, host byte order */
};
```

The routines *getnetbyname*, *getnetbynumber*, and *getnetent* are the network counterparts to the host routines described above. The routines extract their information from *InetDBase:networks*.

Protocol names

For protocols, which are defined in *InetDBase:protocols*, the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*, *getprotobynumber*, and *getprotoent*:

```
struct  protoent {
        char    *p_name;        /* official protocol name */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol number */
};
```

Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific *port* and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher level library routines will have to be bypassed or extended. Services available are contained in the file *InetDBase:services*. A service mapping is described by the *servent* structure:

```
struct  servent {
        char    *s_name;        /* official service name */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port number, network byte order */
        char    *s_proto;       /* protocol to use */
};
```

The routine *getservbyname* maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call:

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a telnet server using any protocol, while the call:

```
sp = getservbyname("telnet", "tcp");
```

returns only that telnet server which uses the TCP protocol. The routines *getservbyport* and *getservent* are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in the example program in *Figure 123.8* on page 5a-338. (This example will be considered in more detail in *Client/server model* on page 5a-339.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be worthwhile.

Aside from the address-related data base routines, there are several other routines available in the Inetlib and Unixlib libraries which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. The table below summarizes the Unixlib routines for manipulating variable length byte strings, and the Inetlib routines for handling byte swapping of network addresses and values:

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, such as ARMs and VAXes, host byte ordering is different than network byte ordering. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

```
printf("port number %d\n", ntohs(sp->s_port));
```

(On machines where unneeded the byte swapping routines are defined as null macros.

Client/server model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server program. This implies an asymmetry in establishing communication between the client and server which has been examined in *Basics* on page 5a-325. In this part of the tutorial we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may

```
#include <stdio.h>
#include "sys/types.h"
#include "sys/socket.h"
#include "netinet/in.h"
#include "netdb.h"
...
main(argc, argv)
        int argc;
        char *argv[];
{
        struct sockaddr_in server;
        struct servent *sp;
        struct hostent *hp;
        int s;
        ...

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }

        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }
        bzero((char *)&server, sizeof (server));
        bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
        server.sin_family = hp->h_addrtype;
        server.sin_port = sp->s_port;

        s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                xperror("rlogin: socket");
                exit(3);
        }
        ...

        /* Connect does the bind() for us */

        if (connect(s, (char *)&server, sizeof (server)) < 0) {
                xperror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 123.8 Remote login client code

be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a ‘client program’ and a ‘server program’. We will first consider the properties of server programs, then client programs.

A server program normally listens at a well known address for service requests. That is, the server program remains dormant until a connection is requested by a client’s connection to the server’s address. At such a time the server program ‘wakes up’ and services the client, performing whatever appropriate actions the client requests of it.

Servers

Most servers are accessed at well known Internet addresses. For example, the BSD UNIX remote login server’s main loop is of the form shown in *Figure 123.9* on page 5a-340. (Although this example is a little strange in not being a RISC OS application, it still contains a number of relevant and useful points.)

The first step taken by the server is to look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogind: tcp/login: unknown service\n");
        exit(1);
}
```

The result of the *getservbyname* call is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location.

The main body of the loop is fairly simple:

```
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len);
        if (g < 0) {
                if (errno != EINTR)
                        xperror("rlogind: accept");
                continue;
        }
        doit(g, &from);
        close(g);
}
```

```
main(argc, argv)
        int argc;
        char *argv[]
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

        sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }
        ...

        sin.sin_port = sp->s_port;
        ...
        f = socket(AF_INET, SOCK_STREAM, 0);
        ...
        if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
                ...
        }
        ...
        listen(f, 5);
        for (;;) {
                int g, len = sizeof (from);

                g = accept(f, (struct sockaddr *)&from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                                xperror("rlogind: accept");
                        continue;
                }
                doit(g, &from);
                close(g);
        }
}
```

Figure 123.9 Remote login server

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted, for example by an Escape. Therefore, the return value from *accept* is checked to insure a connection has actually been established, and an error report is printed if an error has occurred.

With a connection in hand, the server then invokes the main body of the remote login protocol processing. The address of the client is also handed the *doit* routine because it requires it in authenticating clients.

Clients

The client side of the remote login service was shown earlier in *Figure 123.8* on page 5a-338. One can see the separate, asymmetric roles of the client and server clearly in the code. The server is a passive entity, listening for client connections, while the client program is an active entity, initiating a connection when invoked.

Let us consider more closely the steps taken by the client remote login program. As in the server program, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
        fprintf(stderr, "rlogin: tcp/login: unknown service\n");
        exit(1);
}
```

Next the destination host is looked up with a *gethostbyname* call:

```
hp = gethostbyname(argv[1])
if (hp == NULL) {
        fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
}
```

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login program resides on the foreign host:

```
bzero((char *)&server, sizeof (server));
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. Note that *connect* implicitly performs a *bind* call, since *s* is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        xperror("rlogin: socket");
        exit(3);
}
...
if (connect(s, (struct sockaddr *) &server, sizeof (server)) < 0) {
        xperror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

Connectionless servers

While connection-based services are the norm, some services are based on the use of datagram sockets. One, in particular, is the 4.3BSD UNIX ‘rwho’ service which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to *broadcast* information to all hosts connected to a particular network, is of interest as an example usage of datagram sockets.

A user on any machine running the rwho server may find out the current status of a machine with the *ruptime* program. The output generated is illustrated in *Figure 123.10* on page 5a-342.

```
arpa    up      9:45,           5 users, load   1.15,   1.39,   1.31
cad     up      2+12:04,        8 users, load   4.67,   5.13,   4.59
calder  up      10:10,          0 users, load   0.27,   0.15,   0.14
dali    up      2+06:28,        9 users, load   1.04,   1.20,   1.65
degas   up      25+09:48,       0 users, load   1.49,   1.43,   1.41
ear     up      5+00:05,        0 users, load   1.51,   1.54,   1.56
ernie   down    0:24
esvax   down    17:04
ingres  down    0:26
kim     up      3+09:16,        8 users, load   2.03,   2.46,   3.11
matisse up      3+06:18,        0 users, load   0.03,   0.03,   0.05
medea   up      3+09:39,        2 users, load   0.35,   0.37,   0.50
merlin  down    19+15:37
miro    up      1+07:20,        7 users, load   4.59,   3.28,   2.12
monet   up      1+00:43,        2 users, load   0.22,   0.09,   0.07
oz      down    16:09
statvax up      2+15:57,        3 users, load   1.52,   1.81,   1.86
ucbvax  up      9:34,           2 users, load   6.08,   5.16,   3.28
```

Figure 123.10 ruptime output

Status information for each host is periodically broadcast by rwho server programs on each machine. The same server program also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Note that the use of broadcast for such a task is fairly inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

The rwho server, in a simplified form, is pictured in *Figure 123.11* on page 5a-344. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to insure they’ve been sent by another rwho server program, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host

```
main() {
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        on = 1;
        if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on)) < 0) {
                xperror("setsockopt SO_BROADCAST");
                exit(1);
        }
        bind(s, (struct sockaddr *) &sin, sizeof (sin));
        ...
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0,
                    (struct sockaddr *)&from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                xperror("rwhod: recv");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        fprintf(stderr, "rwhod: %d: bad from port",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        fprintf(stderr, "rwhod: malformed host name from
                            %x", ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s",RWHODIR,wd.wd_hostname);
                whod = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}

onalrm() {
        /* Broadcast our status to other rwho servers, and then use
         * OS_CallAfter to re-enter this function after a given interval.
}
```

Figure 123.11 rwho server

has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

The second task performed by the server is to supply information regarding the status of its host. This involves periodically acquiring system status information, packaging it up in a message and broadcasting it on the local network for other rwho servers to hear. The supply function *onalrm* is triggered by a timer, which it sets itself. Locating the system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematical, however.

Status information must be broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbours (based on the status messages received from other rwho servers). This, unfortunately, requires some bootstrapping information, for a server will have no idea what machines are its neighbours until it receives status messages from them. Therefore, if all machines on a net are freshly booted, no machine will have any known neighbours and thus never receive, or send, any status information. This is the identical problem faced by the routing table management program in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbours and request that they always communicate with these neighbours. If each server has at least one neighbour supplied to it, status information may then propagate through a neighbour to hosts which are not (possibly) directly neighbours. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information†.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.3BSD attempts to isolate host-specific information from applications by providing system calls which return the necessary information‡.

A mechanism exists, in the form of a *socketioctl* call, for finding the collection of networks to which a host is directly connected. Further, a local network broadcasting mechanism has been implemented at the socket level. Combining these two features allows a program to broadcast on any directly connected local network which supports

† One must, however, be concerned about ‘loops’. That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

‡ An example of such a system call is the *gethostname* call which returns the host’s ‘official’ name.

the notion of broadcasting in a site independent manner. This allows 4.3BSD to solve the problem of deciding how to propagate status information in the case of *rwho*, or more generally in broadcasting. Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via the appropriate *socketioctl* calls. The specifics of such broadcastings are complex, however, and will be covered in *Broadcasting and determining network configuration* on page 5a-353.

The Internet event

(This description of the Internet event supersedes the old description on page 1-161.)

Under 4.3 BSD, signals are used to notify processes of specific events. Under RISC OS, the Internet event performs a similar function:

Internet event

R0 = 19
R1 = event subcode:
- 1 ⇒ a socket has input waiting to be read
- 2 ⇒ an urgent event has occurred, such as the arrival of out-of-band data
- 3 ⇒ socket connection is broken
- 4 ⇒ a RevARP server has replied to a RevARP request

R2 = socket descriptor (if R1 = 1, 2, or 3), or IP address of replying server (if R1 = 4)
R3 = IP address of requesting station (if R1 = 4)

This event is generated when certain Internet events occur:

```
#define Internet_Event          19

#define Socket_Async_Event      1
#define Socket_Urgent_Event     2
#define Socket_Broken_Event     3
#define RarpReply               4
```

- The event *Internet_Event/Socket_Async_Event* allows an event handler within a program to run when a socket has input waiting to be read; normally the event handler will make a *recv* call to read expected data, or an *accept* call to receive an expected call.
- The event *Internet_Event/Socket_Urgent_Event* allows an event handler to run if some urgent event, such as the arrival of out-of-band data, occurs.
- The event *Internet_Event/Socket_Broken_Event* allows an event handler to run if a socket connection is broken.
- The event *Internet_Event/RarpReply* allows an event handler to run if a RevARP server has replied to a RevARP request.

Note that event subcodes 1, 2 and 3 are approximately equivalent to the UNIX *SIGIO*, *SIGURG* and *SIGPIPE* signals respectively, and are generated under equivalent circumstances.

Using the Internet event

Use of the event facility requires these steps:

1. You must set up an event handler (see *Events* on page 1-147), and then claim the event vector using the SWI OS_Claim (page 1-66).
2. You must enable the Internet event using the SWI OS_Byte 14 (page 1-152).
3. You must make a *socketioctl* FIOASYNC call for every socket that you require to generate the event *Internet_Event/Socket_Async_Event*:

```
/* Allow receipt of asynchronous I/O events */

#include "sys/ioctl.h"
…
int s;
int on = 1;
…
s = socket(AF_INET, SOCK_STREAM, 0);
…
if (socketioctl(s, FIOASYNC, &on) < 0) {
        xperror("socketioctl error");
        return(-1);
}
…
```

   The Internet module only generates this event for a socket once you've made this call.

Advanced topics

A number of facilities have yet to be discussed. For most users of the communication system the mechanisms already described will suffice in constructing distributed applications. However, others will find the need to utilise some of the features which we consider in this section.

Out-of-band data

The stream socket abstraction includes the notion of out-of-band data. Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independently of normal data.

The abstraction defines that the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signalling (ie the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data.

It is possible to 'peek' (via MSG_PEEK) at out-of-band data. The Internet event *Socket_Urgent_Event* (see page 5a-345) is generated when the protocol is notified of its existence. If multiple sockets may have out-of-band data awaiting delivery, a *select* call for exceptional conditions may be used to determine those sockets with such data pending. Neither the event nor the select indicate the actual arrival of the out-of-band data, but only notification that it is pending.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out-of-band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server programs. When a signal flushes any pending output from the remote program(s), all data up to the mark in the data stream is discarded.

To send an out-of-band message the MSG_OOB flag is supplied to a *send* or *sendto* calls, while to receive out-of-band data MSG_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, the SIOCATMARK *socketioctl* is provided:

```
socketioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out-of-band data has arrived), the next read will provide data sent by the client prior to transmission of the out-of-band signal. The routine used in the remote login program to flush output – for example on an Escape – is shown in *Figure 123.12* below. It reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

```
#include "sys/ioctl.h"
#include "sys/file.h"
#include "kernel.h"
#include "swis.h"
...
char waste[BUFSIZ]; /* global rather than auto; doesn't go on SVC stack */

oob() {
        char mark;
        _kernel_swi_regs r;

        for (;;) {
                if (socketioctl(rem, SIOCATMARK, &mark) < 0) {
                        xperror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) socketread(rem, waste, sizeof (waste));
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
                xperror("recv");
                ...
        }
        ...
}
```

Figure 123.12 Flushing I/O on receipt of out-of-band data

A program may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (eg the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a *recv* is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The program must then read enough of the queued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (eg *telnet*) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, SO_OOBINLINE; see *setsockopt* for usage. With this option, the position of urgent data (the 'mark') is retained, but the urgent data immediately follows the mark within the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

Selecting specific protocols

If the third argument to the *socket* call is 0, *socket* will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. However, when using ‘raw’ sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument may be important for setting up demultiplexing. For example, raw sockets in the Internet family may be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol one determines the protocol number as defined within the communication domain. For the Internet domain one may use one of the library routines discussed in section 3, such as *getprotobyname*:

```
#include "sys/types.h"
#include "sys/socket.h"
#include "netinet/in.h"
#include "netdb.h"
...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This would result in a socket *s* using a stream based connection, but with protocol type of ‘newtcp’ instead of the default ‘tcp.’

Address binding

As was mentioned in the earlier section *Basics*, binding addresses to sockets in the Internet domain can be fairly complex. As a brief reminder, these associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the *bind* system call, a program may specify half of an association, the <local address, local port> part, while the *connect* and *accept* primitives are used to complete a socket’s association by specifying the <foreign address, foreign port> part. Since the association is created in two steps the association uniqueness requirement indicated previously could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain the notion of a ‘wildcard’ address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in *"netinet/in.h"*), the system interprets the address as ‘any valid address’. For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include "sys/types.h"
#include "netinet/in.h"
...
struct sockaddr_in sin;
...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Sockets with wildcarded local addresses may receive messages directed to the specified port number, and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the program will be able to accept connection requests which are addressed to 128.32.0.4 or 10.0.0.78. If a server program wished to only allow hosts on a given network connect to it, it would bind the address of the host on the appropriate network.

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
        ...
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The system selects the local port number based on two criteria. The first is that ‘privileged’ Internet ports below IPPORT_RESERVED (1024) must be specifically requested by a program, whereas higher values are used by RISC OS when it chooses a port number, the program not having specified one. The second is that the port number

is not currently bound to some other socket. In order to find a free Internet port number in the privileged range the *rresvport* library routine may be used as follows to return a stream socket with a privileged port number:

```
int lport = IPPORT_RESERVED – 1;
int s;
...
s = rresvport(&lport);
if (s < 0) {
        if (errno == EAGAIN)
                fprintf(stderr, "socket: all ports in use\n");
        else
                xperror("rresvport: socket");
        ...
}
```

The restriction on allocating ports was done to allow programs executing in a ‘secure’ environment to perform authentication based on the originating address and port number. The port number and network address of the machine from which the user is logging in can be determined either by the *from* result of the *accept* call, or from the *getpeername* call.

In certain cases the algorithm used by the system in selecting port numbers is unsuitable for an application. This is because associations are created in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation the system would disallow binding the same local address and port number to a socket if a previous data connection’s socket still existed. To override the default port selection algorithm, an option call must be performed prior to address binding:

```
...
int     on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

With the above call, local addresses may be bound which are already in use. This does not violate the uniqueness requirement as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned.

Broadcasting and determining network configuration

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets which are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: it is desired to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbours.

To send a broadcast message, a datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int     on = 1;

setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

and at least a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in *"netinet/in.h"*). To determine the list of addresses for all reachable neighbours requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, RISC OS provides a method of retrieving this information from the system data structures. The SIOCGIFCONF *socketioctl* call returns the interface configuration of a host in the form

of a single *ifconf* structure; this structure contains a ‘data area’ which is made up of an array of *ifreq* structures, one for each network interface to which the host is connected. These structures are defined in *"net/if.h"* as follows:

```
struct ifconf {
        int     ifc_len;                /* size of associated buffer */
        union {
                caddr_t ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
};

#define ifc_buf ifc_ifcu.ifcu_buf       /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req       /* array of structures returned */

#define IFNAMSIZ        16

struct ifreq {
        char    ifr_name[IFNAMSIZ];     /* if name, eg "en0" */
        union {
                struct  sockaddr ifru_addr;
                struct  sockaddr ifru_dstaddr;
                struct  sockaddr ifru_broadaddr;
                short   ifru_flags;
                caddr_t ifru_data;
        } ifr_ifru;
};

#define ifr_addr        ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define ifr_broadaddr   ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags       ifr_ifru.ifru_flags     /* flags */
#define ifr_data        ifr_ifru.ifru_data      /* for use by interface */
```

The actual call which obtains the interface configuration is

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (socketioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
        ...
}
```

After this call *buf* will contain one *ifreq* structure for each network to which the host is connected, and *ifc.ifc_len* will have been modified to reflect the number of bytes used by the *ifreq* structures.

For each structure there exists a set of ‘interface flags’ which tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The SIOCGIFFLAGS *socketioctl* retrieves these flags for an interface specified by an *ifreq* structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
        /*
         * We must be careful that we don’t use an interface
         * devoted to an address family other than those intended;
         * if we were interested in NS interfaces, the
         * AF_INET would be AF_NS.
         */
        if (ifr->ifr_addr.sa_family != AF_INET)
                continue;
        if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
                ...
        }
        /*
         * Skip boring cases.
         */
        if ((ifr->ifr_flags & IFF_UP) == 0 ||
            (ifr->ifr_flags & IFF_LOOPBACK) ||
            (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
                continue;
```

Once the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks this is done via the SIOCGIFBRDADDR *socketioctl*, while for point-to-point networks the address of the destination host is obtained with SIOCGIFDSTADDR.

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTTOPOINT) {
        if (socketioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
                ...
        }
        bcopy((char *) ifr->ifr_dstaddr, (char *) &dst,
            sizeof (ifr->ifr_dstaddr)); }
else if (ifr->ifr_flags & IFF_BROADCAST) {
        if (socketioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
                ...
        }
        bcopy((char *) ifr->ifr_broadaddr, (char *) &dst,
            sizeof (ifr->ifr_broadaddr));
}
```

After the appropriate *socketioctl*'s have obtained the broadcast or destination address (now in *dst*), the *sendto* call may be used:

```
        sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof (dst));
}
```

In the above loop one *sendto* occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a program only wished to send broadcast messages on a given network, code similar to that outlined above would be used, but the loop would need to find the correct destination address.

Received broadcast messages contain the sender's address and port, as datagram sockets are bound before a message is allowed to go out.

Socket options

It is possible to set and get a number of options on sockets via the *setsockopt* and *getsockopt* system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows: *s* is the socket on which the option is to be applied. *Level* specifies the protocol layer on which the option is to be applied; in most cases this is the 'socket level', indicated by the symbolic constant SOL_SOCKET, defined in *"sys/socket.h"*. The actual option is specified in *optname*, and is a symbolic constant also defined in *"sys/socket.h"*. *Optval* and *optlen* point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For *getsockopt*, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified upon return to indicate the actual amount of storage used.

An example should help clarify things. It is sometimes useful to determine the type (eg stream, datagram, etc) of an existing socket; programs under *inetd* (described below) may need to perform this task. This can be accomplished as follows via the SO_TYPE socket option and the *getsockopt* call:

```
#include "sys/types.h"
#include "sys/socket.h"

int type, size;

size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
        ...
}
```

After the *getsockopt* call, *type* will be set to the value of the socket type, as defined in *"sys/socket.h"*. If, for example, the socket were a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

Multitasking

The examples in this tutorial – and in the earlier *Introductory tutorial* – assume that they are written for a pre-emptive multitasking environment such as Unix. In such cases, it doesn't matter if a call may not return for an arbitrary length of time, as it will not prevent other software from running. However, RISC OS is a co-operative multitasking environment, which relies on a program returning control to the operating system before other programs can run. It is therefore vital that all the calls that your program makes immediately return control to you.

These are the different ways you can do this:

- Before making a call that might block, call *select* with a zero timeout to determine if the socket is ready for the call. If the socket is ready, then make the call. Otherwise give back control to RISC OS, and retry later on.

  Using the *select* call is described in the earlier tutorials; see also its description on page 5a-438.

- Before you first use a socket, mark it as non-blocking. Any call that would otherwise block no longer does so, but instead returns an EWOULDBLOCK error. If you get that error returned, you should give back control to RISC OS, and retry later on.

  See *Non-blocking sockets* on page 5a-358.

- Use the Internet event to receive notification of when data is available on a socket, and an appropriate event handler to handle the resultant I/O – which will not block, since it does not have to wait for data. The event handler must be in a module so that it is paged into memory when the event occurs.

  See *Interrupt driven socket I/O* on page 5a-358, and *The Internet event* on page 5a-345.

Some of the above methods require you to give back control to RISC OS, and retry later on:

- With a desktop application, you do so by calling Wimp_Poll; however, there is no guarantee how long it will be until control returns to your application.
- An alternative is to use OS_CallAfter or OS_CallEvery to arrange for an address to be called after a given time delay; in this case, the address must be within a module so that it is paged in when called.

Non-blocking sockets

When writing modules, or programs to run under the Wimp, you may often find it convenient to make use of sockets which do not block. That is, I/O requests which cannot complete immediately and would therefore cause the program to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the *socket* call, it may be marked as non-blocking by *socketioctl* as follows:

```
#include "sys/ioctl.h"
…
int s;
int on = 1;
…
s = socket(AF_INET, SOCK_STREAM, 0);
…
if (socketioctl(s, FIONBIO, &on) < 0) {
        xperror("socketioctl");
        return(-1);
}
…
```

When performing non-blocking I/O on sockets, one must be careful to check for the error EWOULDBLOCK (stored in the global variable *errno*), which occurs when an operation would normally block, but the socket it was performed on is marked as non-blocking. In particular, *accept*, *connect*, *send*, *recv*, *read*, and *write* can all return EWOULDBLOCK, and programs should be prepared to deal with such return codes. If an operation such as a *send* cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately will be processed, and the return value will indicate the amount actually sent.

Interrupt driven socket I/O

The event *Internet_Event/Socket_Async_Event* allows a program to be notified via an event when a socket has data waiting to be read. The steps required to use the *Socket_Async_Event* facility are described in *The Internet event* on page 5a-345.

Sample code to allow a given program to receive information on pending I/O requests as they occur for a socket *s* is given in *Figure 123.13* on page 5a-359. With the addition of code to the handler to process the *Socket_Urgent_Event* event subcode, this code can also be used to prepare for receipt of *Internet_Event/Socket_Urgent_Event* events.

```
#include "kernel.h"
#include "swis.h"

main(char *argv, int argc)
{
        if (claim_eventv())
                exit(1); /* Failed immediately, so nothing to tidy */
        if (event_enable()) {
                disable_release_eventv();       /* Release events etc */
                exit(2);
        }
        /* Event handler now installed and working */

        ...
        disable_release_eventv(); /* On exit */
        exit(0)
}

static _kernel_oserror *claim_eventv(void)
{
        _kernel_swi_regs r;
        r.r[0] = EventV;
        r.r[1] = (int)event_entry_name; /*entry veneer compiled by CMHG*/
        r.r[2] = (int)module_wsp;
        return (_kernel_swi(XOS_Bit | OS_Claim, &r, &r));
}

static _kernel_oserror *event_enable(void)
{
        _kernel_swi_regs r;
        r.r[0] = Event_Enable;
        r.r[1] = Internet_Event;
        return (_kernel_swi(XOS_Bit | OS_Byte, &r, &r));
}

static void disable_release_eventv(void)
{
        _kernel_swi_regs r;
        r.r[0] = Event_Disable;
        r.r[1] = Internet_Event;
        (void) _kernel_swi(OS_Byte, &r, &r);

        r.r[0] = EventV;
        r.r[1] = (int)event_entry_name; /*entry veneer compiled by CMHG*/
        r.r[2] = (int)module_wsp;
        (void) _kernel_swi(XOS_Bit | OS_Release, &r, &r);

        return;
}
```

Figure 123.13 Use of asynchronous notification of I/O requests

```
int Internet_event_handler(_kernel_swi_regs *r, void *pw)
{
/*
 * cmhg event handler, for which event_entry_name is the veneer function
 *
 * Parameters:
 *              r       : pointer to registers block
 *              pw : "R12" value established by module initialisation
 * Returns:
 *              0 => interrupt "claimed"
 *              !0 => interrupt not "claimed"
 */

        UNUSED(pw);

        /* cmhg will only pass through this event anyway */
        if (r->r[0] == Internet_Event)
        {
                /* if notification of asynchronous I/O */
                if (r->r[1] == Socket_Async_Event &&
                   (r->r[2] == my_atpsock || r->r[2] == my_routedsock))
                {
                        process_input(r->r[2]);
                        return 0;
                }
        }
        return 1;
}

static void process_input(int sock)
{
/*
 * Process input on a socket: event has been received to indicate I/O is
 * "available" on this socket
 */
        ...
}
```

Figure 123.13 Use of asynchronous notification of I/O requests (continued)

Protocols

ICMP

Name

ICMP – Internet Control Message Protocol

Synopsis

```
#include "sys/socket.h"

int socket(AF_INET, SOCK_RAW, proto);
int proto;
```

Description

ICMP is the error and control message protocol used by IP and the Internet protocol family. It may be accessed through a ‘raw socket’ for network monitoring and diagnostic functions. The *proto* parameter to the socket call to create an ICMP socket is obtained from *getprotobyname*. ICMP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls, though the *connect* call may also be used to fix the destination for future packets (in which case the *recv* and *send* system calls may be used).

Outgoing packets automatically have an IP header prepended to them (based on the destination address). Incoming packets are received with the IP header and options intact.

A socket operation may fail with one of the following errors returned:

[EISCONN]	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
[ENOTCONN]	when trying to send a datagram, but no destination address is specified, and the socket hasn’t been connected;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

IP

Name

IP – Internet Protocol

Synopsis

```
#include "sys/socket.h"

int socket(AF_INET, SOCK_RAW, proto);
int proto;
```

Description

IP is the transport layer protocol used by the Internet protocol family. Options may be set at the IP level when using higher-level protocols that are based on IP (such as TCP and UDP). It may also be accessed through a ‘raw socket’ when developing new protocols, or special purpose applications.

A single generic option is supported at the IP level, IP_OPTIONS, that may be used to provide IP options to be transmitted in the IP header of each outgoing packet. Options are set with *setsockopt* and examined with *getsockopt*. The format of IP options to be sent is that specified by the IP protocol specification, with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use. IP options may be used with any socket type in the Internet family.

Raw IP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls, though the *connect* call may also be used to fix the destination for future packets (in which case the *recv* and *send* system calls may be used).

If *proto* is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If *proto* is non-zero, that protocol number will be used on outgoing packets and to filter incoming packets.

If *proto* is IPPROTO_RAW (or 0, which defaults to that) outgoing packets do not have an IP header prepended to them, but go out ‘as is’. Otherwise outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number the socket is created with). Incoming packets are received with IP header and options intact.

A socket operation may fail with one of the following errors returned:

[EISCONN]	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
[ENOTCONN]	when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

The following errors specific to IP may occur when setting or getting IP options:

[EINVAL]	an unknown socket option name was given;
[EINVAL]	the IP option field was improperly formed; an option field was shorter than the minimum value or longer than the option buffer provided.

TCP

Name

TCP – Internet Transmission Control Protocol

Synopsis

```
#include "sys/socket.h"
int socket(AF_INET, SOCK_STREAM, 0);
```

Description

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. TCP uses the standard Internet address format and, in addition, provides a per-host collection of ‘port addresses’. Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets utilising the tcp protocol are either ‘active’ or ‘passive’. Active sockets initiate connections to passive sockets. By default TCP sockets are created active; to create a passive socket the *listen* socket call must be used after binding the socket with the *bind* system call. Only passive sockets may use the *accept* call to accept incoming connections. Only active sockets may use the *connect* call to initiate connections.

Passive sockets may ‘underspecify’ their location to match incoming connection requests from multiple networks. This technique, termed ‘wildcard addressing’, allows a single server to provide service to clients on multiple networks. To create a socket which listens on all networks, the Internet address INADDR_ANY must be bound. The TCP port may still be specified at this time; if the port is not specified the system will assign one. Once a connection has been established the socket’s address is fixed by the peer entity’s location. The address assigned to the socket is the address associated with the network interface through which packets are being transmitted and received. Normally this address corresponds to the peer entity’s network.

TCP supports one socket option which is set with *setsockopt* and tested with *getsockopt*. Under most circumstances, TCP sends data when it is presented; when outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgement is received. For a small number of clients, such as

window systems that send a stream of mouse events which receive no replies, this packetisation may cause significant delays. Therefore, TCP provides a boolean option, TCP_NODELAY, to defeat this algorithm. The option level for the *setsockopt* call is the protocol number for TCP, available from *getprotobyname*.

Options at the IP transport level may be used with TCP. Incoming connection requests that are source-routed are noted, and the reverse source route is used in responding.

A socket operation may fail with one of the following errors returned:

[EISCONN]	when trying to establish a connection on a socket which already has one;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[ETIMEDOUT]	when a connection was dropped due to excessive retransmissions;
[ECONNRESET]	when the remote peer forces the connection to be closed;
[ECONNREFUSED]	when the remote peer actively refuses connection establishment (usually because no program is listening to the port);
[EADDRINUSE]	when an attempt is made to create a socket with a port which has already been allocated;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

UDP

Name

UDP – Internet User Datagram Protocol

Synopsis

```
#include "sys/socket.h"
int socket(AF_INET, SOCK_DGRAM, 0);
```

Description

UDP is a simple, unreliable datagram protocol which is used to support the SOCK_DGRAM abstraction for the Internet protocol family. UDP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls, though the connect call may also be used to fix the destination for future packets (in which case the *recv* and *send* system calls may be used).

UDP address formats are identical to those used by TCP. In particular UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space (ie a UDP port may not be ‘connected’ to a TCP port). In addition broadcast packets may be sent (assuming the underlying network supports this) by using a reserved ‘broadcast address’; this address is network interface dependent.

Options at the IP transport level may be used with UDP.

A socket operation may fail with one of the following errors returned:

[EISCONN]	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
[ENOTCONN]	when trying to send a datagram, but no destination address is specified, and the socket hasn’t been connected;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[EADDRINUSE]	when an attempt is made to create a socket with a port which has already been allocated;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

Library calls

INDEX

The following symbols are exported by Socklib, Inetlib and Unixlib:

Symbol	from	See	on page
socketwritev	Socklib	*SOCKETWRITE*	5a-452
strcasecmp	Unixlib	*STRING*	5a-454
strncasecmp	Unixlib	*STRING*	5a-454
sys_errlist	Unixlib	*XPERROR*	5a-459
sys_nerr	Unixlib	*XPERROR*	5a-459
unlink	Unixlib	*UNLINK*	5a-455
_varnamebuf	Unixlib	*_VARNAMEBUF*	5a-456
write	Unixlib	*WRITE*	5a-457
writev	Unixlib	*WRITE*	5a-457
xgets	Unixlib	*XGETS*	5a-458
xperror	Unixlib	*XPERROR*	5a-459
xputchar	Unixlib	*XPUTCHAR*	5a-460

ACCEPT

Name

accept – accept a connection on a socket

Synopsis

```
#include "sys/socket.h"
#include "sys/types.h"

int accept(s, addr, addrlen)
int s;
struct sockaddr *addr;
int *addrlen;
```

Description

The argument *s* is a socket that has been created with *socket*, bound to an address with *bind*, and is listening for connections after a *listen*. *Accept* extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s*, and allocates a new socket descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *accept* blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, *accept* returns an error as described below. The accepted socket may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring (eg Internet). The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

Return value

The call returns –1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

Errors

The call will fail if:

[EBADF]	The descriptor is invalid.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The *addr* parameter is invalid.
[EWOULDBLOCK]	The socket is marked non-blocking and no connections are present to be accepted.

See also

bind (page 5a-375), connect (page 5a-382), listen (page 5a-424), select (page 5a-438), socket (page 5a-443)

Exported by

Socklib

ACCESS

Name

access – determine accessibility of file

Synopsis

```
#include "sys/fcntl.h"

#define R_OK 4  /* test for read permission */
#define W_OK 2  /* test for write permission */
#define X_OK 1  /* execute permission, ignored */
#define F_OK 0  /* test for presence of file */

int access(path, mode)
char *path;
int mode;
```

Description

Access checks the given file *path* for accessibility according to *mode*, which is an inclusive or of the bits R_OK, W_OK and X_OK, defined in *sys/fcntl.h*. Specifying *mode* as F_OK (ie 0) tests whether the directories leading to the file can be searched and the file exists.

Notice that only access bits are checked. A directory may be indicated as writable by *access*, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but executing it will fail unless it is in proper format.

Return value

If *path* cannot be found or if any of the desired access modes would not be granted, then a –1 value is returned; otherwise a 0 value is returned.

Errors

Access to the file is denied if one or more of the following are true:

[ENOENT]	The named file does not exist.
[EACCES]	Permission bits of the file mode do not permit the requested access. The permission is checked with respect to the ‘owner’ read and write mode bits.

See also

chmod (page 5a-380), filestat (page 5a-388)

Exported by

Unixlib

BIND

Name

bind – bind a name to a socket

Synopsis

```
#include "sys/socket.h"
#include "sys/types.h"

int bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

Description

Bind assigns a name to an unnamed socket. When a socket is created with *socket* it exists in a name space (address family) but has no name assigned. *Bind* requests that *name* be assigned to the socket.

The rules used in name binding vary between communication domains.

Return value

If the bind is successful, a 0 value is returned. A return value of –1 indicates an error, which is further specified in the global *errno*.

Errors

The call will fail if:

[EBADF]	*s* is not a valid descriptor.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EFAULT]	The name parameter is invalid.

See also

connect (page 5a-382), listen (page 5a-424), socket (page 5a-443), getsockname (page 5a-409)

Exported by

Socklib

BSTRING

Name

bcmp, bcopy, bzero – byte string operations

Synopsis

```
void bcopy(src, dst, length)
char *src, *dst;
int length;

int bcmp(b1, b2, length)
char *b1, *b2;
int length;

char *bzero(b, length)
char *b;
int length;
```

Description

The functions *bcopy*, *bcmp* and *bzero* operate on variable length strings of bytes. They do not check for null bytes as the routines in *string* do.

Bcopy copies *length* bytes from string *src* to the string *dst*.

Bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

Bzero places *length* null (0) bytes in the string *b*.

Exported by

Unixlib

BYTEORDER

Name

htonl, htons, ntohl, ntohs – convert values between host and network byte order

Synopsis

```
int htons(hostshort);
int hostshort;

int ntohs(netshort);
int netshort;

#include "sys/types.h"

u_long htonl(hostlong);
u_long hostlong;

u_long ntohl(netlong);
u_long netlong;
```

Description

These routines convert 16 and 32 bit quantities between network byte order and host byte order.

These routines are most often used in conjunction with Internet addresses and ports as returned by *gethostbyname* and *getservent*.

See also

gethostbyname (page 5a-394), getservent (page 5a-407)

Exported by

Inetlib

CHDIR

Name

chdir – change current working directory

Synopsis

```
int chdir(path)
char *path;
```

Description

Path is the pathname of a directory. *Chdir* causes this directory to become the current working directory, the starting point for incomplete path names. If *path* specifies a different filing system, it also selects that as the current filing system. If *path* is a null string, the directory is set to the user root directory.

Return value

Upon completion, a value of 0 is returned.

Errors

Chdir will fail and the current working directory will be unchanged if the named directory does not exist.

Exported by

Unixlib

CHMOD

Name

chmod – change mode of file

Synopsis

```
int chmod(path, mode)
char *path;
int mode;
```

Description

The file whose name is given by *path* has its read and write attributes changed to those in *mode*. Modes are constructed by *or*'ing together some combination of the following:

IREAD	00400	read by owner
IWRITE	00200	write by owner

Other bits acted on by the Unix version of this command are ignored.

Return value

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Errors

Chmod will fail and the file mode will be unchanged if:

[ENOENT]	The named file does not exist.

See also

access (page 5a-373)

Exported by

Unixlib

CLOSE

Name

close – delete a descriptor

Synopsis

```
int close(d)
int d;
```

Description

Close is a synonym for *socketclose*; see page 5a-446. The call is provided mainly so that you do not need to rename *close* calls in code that you are porting.

See also

socketclose (page 5a-446)

Exported by

Unixlib

CONNECT

Name

connect – initiate a connection on a socket

Synopsis

```
#include "sys/socket.h"
#include "sys/types.h"

int connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

Description

The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name*, which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way. Generally, stream sockets may successfully *connect* only once; datagram sockets may use *connect* multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

Return value

If the connection or binding succeeds, then 0 is returned. Otherwise a –1 is returned, and a more specific error code is stored in *errno*.

Errors

The call fails if:

[EBADF]	*s* is not a valid descriptor.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.

[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network isn't reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter was invalid.
[EINPROGRESS]	The socket is non-blocking and the connection cannot be completed immediately.
[EALREADY]	The socket is non-blocking and a previous connection attempt has not yet been completed.

See also

accept (page 5a-371), select (page 5a-438), socket (page 5a-443), getsockname (page 5a-409)

Exported by

Socklib

ERRNO

Name

errno – global error variable

Synopsis

```
int errno;
```

Description

The global error variable *errno* is used by several libraries – including Socklib – to provide diagnostics for errors when making calls. Typically, when an error occurs the call returns –1, and *errno* is set to a value that indicates the reason for the error. Possible values *errno* may take are:

Value	Name	Meaning
0		Error 0
1	EPERM	Not owner
2	ENOENT	No such file or directory
3	ESRCH	No such process
4	EINTR	Interrupted system call
5	EIO	I/O error
6	ENXIO	No such device or address
7	E2BIG	Arg list too long
8	ENOEXEC	Exec format error
9	EBADF	Bad file number
10	ECHILD	No children
11	EAGAIN	Resource temporarily unavailable
12	ENOMEM	Not enough memory
13	EACCES	Permission denied
14	EFAULT	Bad address
15	ENOTBLK	Block device required
16	EBUSY	Device busy
17	EEXIST	File exists
18	EXDEV	Cross-device link
19	ENODEV	No such device
20	ENOTDIR	Not a directory

Value	Name	Meaning
21	EISDIR	Is a directory
22	EINVAL	Invalid argument
23	ENFILE	File table overflow
24	EMFILE	Too many open files
25	ENOTTY	Inappropriate I/O control operation
26	ETXTBSY	Text file busy
27	EFBIG	File too large
28	ENOSPC	No space left on device
29	ESPIPE	Illegal seek
30	EROFS	Read-only file system
31	EMLINK	Too many links
32	EPIPE	Broken pipe
33	EDOM	Argument value error
34	ERANGE	Result out of range
35	EWOULDBLOCK	Operation would block
36	EINPROGRESS	Operation now in progress
37	EALREADY	Operation already in progress
38	ENOTSOCK	Socket operation on non-socket
39	EDESTADDRREQ	Destination address required
40	EMSGSIZE	Message too long
41	EPROTOTYPE	Protocol wrong type for socket
42	ENOPROTOOPT	Option not supported by protocol
43	EPROTONOSUPPORT	Protocol not supported
44	ESOCKTNOSUPPORT	Socket type not supported
45	EOPNOTSUPP	Operation not supported on socket
46	EPFNOSUPPORT	Protocol family not supported
47	EAFNOSUPPORT	Address family not supported by protocol family
48	EADDRINUSE	Address already in use
49	EADDRNOTAVAIL	Can’t assign requested address
50	ENETDOWN	Network is down
51	ENETUNREACH	Network is unreachable
52	ENETRESET	Network dropped connection on reset
53	ECONNABORTED	Software caused connection abort

Value	Name	Meaning
54	ECONNRESET	Connection reset by peer
55	ENOBUFS	No buffer space available
56	EISCONN	Socket is already connected
57	ENOTCONN	Socket is not connected
58	ESHUTDOWN	Can’t send after socket shutdown
59	ETOOMANYREFS	Too many references: can’t splice
60	ETIMEDOUT	Connection timed out
61	EREFUSED	Connection refused
62	ELOOP	Too many levels of symbolic links
63	ENAMETOOLONG	File name too long
64	EHOSTDOWN	Host is down
65	EHOSTUNREACH	Host is unreachable
66	ENOTEMPTY	Directory not empty
67	EPROCLIM	Too many processes
68	EUSERS	Too many users
69	EDQUOT	Disc quota exceeded
70	ESTALE	Stale NFS file handle
71	EREMOTE	Too many levels of remote in path
72	ENOSTR	Not a stream device
73	ETIME	Timer expired
74	ENOSR	Out of stream resources
75	ENOMSG	No message of desired type
76	EBADMSG	Not a data message
77	EIDRM	Identifier removed
78	EDEADLK	Deadlock situation detected/avoided
79	ENOLCK	No record locks available
80	ENOMSG	No suitable message on queue
81	EIDRM	Identifier removed from system
82	ELIBVER	Wrong version of shared library
83	ELIBACC	Permission denied (shared library)
84	ELIBLIM	Shared libraries nested too deeply
85	ELIBNOENT	Shared library file not found
86	ELIBNOEXEC	Shared library exec format error
87	ENOSYS	Function not implemented

For details of the errors individual calls may return, see their documentation.

See also

_inet_error (page 5a-421), xperror (page 5a-459)

Exported by

Socklib

FILESTAT

Name

filestat – get file status

Synopsis

```
int filestat(path, type)
char *path;
char *type;
```

Description

Filestat obtains information about the file *path*. Read or write permission of the named file is not required, but all directories listed in the path name leading to the file must be reachable. The file is searched for using the path held in the RISC OS system variable *File$Path*. If *path* contains wildcards, only the first file matching the wildcard specification is read.

On exit, *type* contains the file's object type:

0	Not found
1	File found
2	Directory found
3	Image file found (ie both file and directory)

Return value

Upon successful completion the length of the file is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Errors

Filestat will fail if:

[ENOENT]	The named file does not exist

See also

access (page 5a-373)

Exported by

Unixlib

FLUSHINPUT

Name

flushinput – flushes the input buffer

Synopsis

```
void flushinput()
```

Description

Flushinput flushes the current RISC OS input buffer. The contents of the buffer are discarded.

Exported by

Unixlib

FSTAT

Name

fstat – get socket status

Synopsis

```
int fstat(sd, buf)
int sd;
char *buf;
```

Description

Fstat is a synonym for *socketstat*; see page 5a-450. The call is provided mainly so that you do not need to rename *fstat* calls in code that you are porting.

See also

socketstat (page 5a-450)

Exported by

Unixlib

GETDTABLESIZE

Name

getdtablesize – get descriptor table size

Synopsis

```
int getdtablesize()
```

Description

Getdtablesize is a synonym for *getstablesize*; see page 5a-413. The call is provided mainly so that you do not need to rename *getdtablesize* calls in code that you are porting.

See also

getstablesize (page 5a-413)

Exported by

Unixlib

GETEGID

Name

getegid – get group identity

Synopsis

```
int getegid()
```

Description

Getegid returns the effective group ID of the current process.

As RISC OS has no concept of group IDs, the Unixlib version of this call always returns 9999. The call is provided mainly so that you do not need to remove calls to *getegid* from code that you are porting.

See also

getuid (page 5a-415)

Exported by

Unixlib

GETGROUPS

Name

getgroups – get group access list

Synopsis

```
int getgroups(gidsetlen, gidset)
int gidsetlen, *gidset;
```

Description

Getgroups gets the current group access list of the user process and stores it in the array *gidset*. The parameter *gidsetlen* indicates the number of entries that may be placed in *gidset. Getgroups* returns the actual number of groups returned in *gidset*. No more than NGROUPS, as defined in "*sys/param.h*", will ever be returned.

Note that the *gidset* array should be of type `gid_t`, but remains integer for compatibility with earlier BSD Unix systems.

As RISC OS has no concept of group access lists, the Unixlib version of this call always places the single group ID 9999 in the array *gidset*, and returns 1. The call is provided mainly so that you do not need to remove calls to *getgroups* from code that you are porting.

Return value

This call always returns 1, which is the number of groups in the group set.

Exported by

Unixlib

GETHOSTBYNAME

Name

gethostbyname, gethostbyaddr, gethostent, sethostent, endhostent – get network host entry

Synopsis

```
void sethostent(stayopen)
int stayopen;

void endhostent()

#include "netdb.h"

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr;
int len, type;

struct hostent *gethostent()
```

Description

Gethostbyname and *gethostbyaddr* each return a pointer to an object describing an Internet host referenced by name or by address, respectively. The calls query entries in a local data base file, setup as *InetDBase:hosts*. The information is returned in the following structure:

```
struct hostent {
        char    *h_name;
        char    **h_aliases;
        int     h_addrtype;
        int     h_length;
        char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

The members of this structure are:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A zero terminated array of network addresses for the host. Host addresses are returned in network byte order.
h_addr	The first address in h_addr_list.

Gethostent reads the next line of *InetDBase:hosts*, opening the file if necessary.

Sethostent opens and rewinds the file. If the *stayopen* argument is non-zero, the hosts data base will not be closed after each call to *gethostbyname* or *gethostbyaddr*.

Endhostent closes the file.

The *_host_stayopen* symbol is exported for internal use only. You must not use it in your own code.

Return value

Error return status from *gethostbyname* and *gethostbyaddr* is indicated by return of a null pointer.

Bugs

All information is contained in a static area so it must be copied if it is to be saved.

Exported by

Inetlib

GETHOSTNAME

Name

gethostname – get name of current host

Synopsis

```
int gethostname(name, namelen)
char *name;
int namelen;
```

Description

Gethostname returns the standard Internet host name for the current processor, as set in the system variable *Inet$HostName*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

If the system variable *Inet$HostName* is not set, or if it is set to the null string, then the call attempts to set it to 'ARM_NoName', and – whether or not successful – this is also the name returned in the *name* array.

Return value

A zero value is always returned.

Bugs

Host names are limited to MAXHOSTNAMELEN (from *"sys/param.h"*) characters, currently 64.

All information is contained in a static area so it must be copied if it is to be saved.

Exported by

Unixlib

GETLOGIN

Name

getlogin – get login name

Synopsis

```
char *getlogin()
```

Description

Getlogin is a synonym for *getvarusername*; see page 5a-416. The call is provided mainly so that you do not need to rename *getlogin* calls in code that you are porting.

See also

getvarusername (page 5a-416)

Exported by

Unixlib

GETNETENT

Name

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

Synopsis

```
void setnetent(stayopen)
int stayopen;

void endnetent()

#include "netdb.h"

struct netent *getnetent()

struct netent *getnetbyname(name)
char *name;

struct netent *getnetbyaddr(net, type)
int net, type;
```

Description

Getnetent, *getnetbyname*, and *getnetbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, *InetDBase:networks*.

```
struct netent {
        char            *n_name;
        char            **n_aliases;
        int             n_addrtype;
        unsigned long   n_net;
};
```

The members of this structure are:

n_name	The official name of the network.
n_aliases	A zero terminated list of alternate names for the network.
n_addrtype	The type of the network number returned; currently only AF_INET.
n_net	The network number. Network numbers are returned in machine byte order.

Getnetent reads the next line of *InetDBase:networks*, opening the file if necessary.

Setnetent opens and rewinds the file. If the *stayopen* argument is non-zero, the net data base will not be closed after each call to *getnetbyname* or *getnetbyaddr.*

Endnetent closes the file.

Getnetbyname and *getnetbyaddr* sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

The *_net_stayopen* symbol is exported for internal use only. You must not use it in your own code.

Return value

A Null pointer (0) is returned on EOF or error.

Bugs

All information is contained in a static area so it must be copied if it is to be saved.

Exported by

Inetlib

GETPASS

Name

getpass – read a password

Synopsis

```
char *getpass(prompt)
char *prompt;
```

Description

Getpass reads a password from the current input stream, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters.

Bugs

The return value points to static data whose content is overwritten by each call.

Exported by

Unixlib

GETPEERNAME

Name

getpeername – get name of connected peer

Synopsis

```
#include "sys/socket.h"
#include "sys/types.h"

int getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Description

Getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

Return value

A 0 is returned if the call succeeds, –1 if it fails.

Errors

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter was invalid.

See also

accept (page 5a-371), bind (page 5a-375), socket (page 5a-443), getsockname (page 5a-409)

Exported by

Socklib

GETPID

Name

getpid – get process identification

Synopsis

```
int getpid()
```

Description

Getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

As RISC OS has no concept of process IDs, the Unixlib version of this call always returns 9999. The call is provided mainly so that you do not need to remove calls to *getpid* from code that you are porting.

Exported by

Unixlib

GETPROTOENT

Name

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

Synopsis

```
void setprotoent(stayopen)
int stayopen;

void endprotoent()

#include "netdb.h"

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;
```

Description

Getprotoent, *getprotobyname*, and *getprotobynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the Internet protocol data base, *InetDBase:protocols*.

```
struct protoent {
        char    *p_name;
        char    **p_aliases;
        int     p_proto;
};
```

The members of this structure are:

p_name	The official name of the protocol.
p_aliases	A zero terminated list of alternate names for the protocol.
p_proto	The protocol number.

Getprotoent reads the next line of *InetDBase:protocols*, opening the file if necessary.

Setprotoent opens and rewinds the file. If the *stayopen* argument is non-zero, the protocol data base will not be closed after each call to *getprotobyname*.

Endprotoent closes the file.

Getprotobyname and *getprotobynumber* sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

The *_proto_stayopen* symbol is exported for internal use only. You must not use it in your own code.

Return value

A Null pointer (0) is returned on EOF or error.

Bugs

All information is contained in a static area so it must be copied if it is to be saved.

See also

Protocols on page 5a-361

Exported by

Inetlib

GETPWENT

Name

getpwent, getpwuid, getpwnam, setpwent, endpwent – get password file entry

Synopsis

```
void setpwent()

void endpwent()

#include "pwd.h"

struct passwd *getpwuid(uid)
int uid;

struct passwd *getpwnam(name)
char *name;

struct passwd *getpwent()
```

Description

Getpwent, *getpwuid* and *getpwnam* each return a pointer to an object with the following structure.

```
struct  passwd {                /* see getpwent(3) */
        char    *pw_name;
        char    *pw_passwd;
        union { uid_t _uid; int _pad1; } _uid;
        union { gid_t _gid; int _pad2; } _gid;
        int     pw_quota;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};
#define pw_uid  _uid._uid
#define pw_gid  _gid._gid
```

The fields *pw_passwd*, *pw_quota*, *pw_comment*, *pw_gecos*, *pw_dir* and *pw_shell* are unused.

Getpwuid sets *pw_name* to the name returned by *getvarusername*, or to ‘root’ if none is returned; and it sets *pw_uid* to 32767, and *pw_gid* to 9999.

Getpwnam sets *pw_name* to *name*, *pw_uid* to 32767, and *pw_gid* to 9999.

Getpwent does the same as *getpwuid* the first time it is ever called, and the first time it is called after a call to *setpwent* or *endpwent*. It otherwise returns a NULL pointer (0).

Setpwent and *endpwent* have no effect other than altering the behaviour of *getpwent* (see above).

These calls are provided mainly so that you do not need to remove them from code that you are porting.

Bugs

All information is contained in a static area so it must be copied if it is to be saved.

See also

getlogin (page 5a-397), getvarusername (page 5a-416)

Exported by

Unixlib

GETSERVENT

Name

getservent, getservbyport, getservbyname, setservent, endservent – get service entry

Synopsis

```
void setservent(stayopen)
int stayopen;

void endservent()

#include "netdb.h"

struct servent *getservent()

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port;
char *proto;
```

Description

Getservent, *getservbyname*, and *getservbyport* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, *InetDBase:services*.

```
struct servent {
        char    *s_name;
        char    **s_aliases;
        int     s_port;
        char    *s_proto;
};
```

The members of this structure are:

s_name	The official name of the service.
s_aliases	A zero terminated list of alternate names for the service.
s_port	The port number at which the service resides. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

Getservent reads the next line of the file, opening the file if necessary.

Setservent opens and rewinds the file. If the *stayopen* argument is non-zero, the services data base will not be closed after each call to *getservbyname* or *getservbyport*.

Endservent closes the file.

Getservbyname and *getservbyport* sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

The *_serv_stayopen* symbol is exported for internal use only. You must not use it in your own code.

Return value

A Null pointer (0) is returned on EOF or error.

Bugs

All information is contained in a static area so it must be copied if it is to be saved.

See also

getprotoent (page 5a-403)

Exported by

Inetlib

GETSOCKNAME

Name

getsockname – get socket name

Synopsis

```
#include "sys/socket.h"
#include "sys/types.h"

int getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Description

Getsockname returns the current *name* for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

Return value

A 0 is returned if the call succeeds, –1 if it fails.

Errors

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter was invalid.

See also

bind (page 5a-375), socket (page 5a-443)

Exported by

Socklib

GETSOCKOPT

Name

getsockopt, setsockopt – get and set options on sockets

Synopsis

```
int getsockopt(s, level, optname, optval, optlen)
int s, level, optname;
void *optval;
int *optlen;

int setsockopt(s, level, optname, optval, optlen)
int s, level, optname;
void *optval;
int optlen;
```

Description

Getsockopt and *setsockopt* manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost ‘socket’ level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the ‘socket’ level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP (see *GETPROTOENT* on page 5a-403).

The parameters *optval* and *optlen* are used to access option values for *setsockopt*. For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be supplied as 0.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file *"sys/socket.h"* contains definitions for ‘socket’ level options, described below. Options at other protocol levels vary in format and name.

Most socket-level options take an *int* parameter for *optval*. For *setsockopt*, the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a *struct linger* parameter, defined in *"sys/socket.h"*, which specifies the desired state of the option and the linger interval (see below).

The following options are recognized at the socket level. Except as noted, each may be examined with *getsockopt* and set with *setsockopt*.

SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind* call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and programs using the socket are notified via an *Internet_Event/Socket_Broken_Event* event, provided they have enabled it (see *The Internet event* on page 5a-345). SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a *socketclose* is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block on the *socketclose* attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *socketclose* is issued, the system will process the socketclose in a manner that allows control to return to the caller as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with *recv* calls without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume

connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values. Finally, SO_TYPE and SO_ERROR are options used only with *setsockopt*. SO_TYPE returns the type of the socket, such as SOCK_STREAM. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

Return value

A 0 is returned if the call succeeds, –1 if it fails.

Errors

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOPROTOOPT]	The option is unknown at the level indicated.
[EFAULT]	The address pointed to by *optval* is invalid. For *getsockopt*, this error may also be returned if *optlen* is invalid.

See also

ioctl (page 5a-422), socketioctl (page 5a-447), socket (page 5a-443), getprotoent (page 5a-403)

Exported by

Socklib

GETSTABLESIZE

Name

getstablesize – get descriptor table size

Synopsis

```
int getstablesize()
```

Description

The Internet module has a fixed size descriptor table, which is guaranteed to have at least 96 slots. The entries in the descriptor table are numbered with small integers starting at 0. The call *getstablesize* returns the size of this table.

See also

getdtablesize (page 5a-391), close (page 5a-381), socketclose (page 5a-446), select (page 5a-438)

Exported by

Socklib

GETTIMEOFDAY

Name

gettimeofday – get date and time

Synopsis

```
#include "sys/time.h"

int gettimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;
```

Description

The system's notion of the current Greenwich time and the current time zone is obtained with the *gettimeofday* call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. If *tzp* is zero, the time zone information will not be returned or set.

The structures pointed to by *tp* and *tzp* are defined in *"sys/time.h"* as:

```
struct timeval {
        long    tv_sec;         /* seconds since Jan. 1, 1970 */
        long    tv_usec;        /* and microseconds */
};

struct timezone {
        int     tz_minuteswest; /* of Greenwich */
        int     tz_dsttime;     /* type of dst correction to apply */
};
```

The *timezone* structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

Return value

A zero value is always returned. If the date is unset or out of the representable range, then *tv_sec* is –1.

Exported by

Unixlib

GETUID

Name

getuid, geteuid – get user identity

Synopsis

```
int getuid()
int geteuid()
```

Description

Getuid returns the real user ID of the current process, *geteuid* the effective user ID.

As RISC OS has no concept of user IDs, the Unixlib version of this call always returns 32767. The call is provided mainly so that you do not need to remove calls to *getuid* and *geteuid* from code that you are porting.

See also

getegid (page 5a-392)

Exported by

Unixlib

GETVAR

Name

getvarhostname, getvarusername – get host and user names from system variables

Synopsis

```
char *getvarhostname()
char *getvarusername()
```

Description

Getvarhostname returns the standard Internet host name for the current processor, as set in the system variable *Inet$HostName*. If the variable is not set, or if it is set to the null string, then the call first attempts to set it to ‘ARM_NoName’.

Getvarusername returns the current user name, as previously set in the system variable *Inet$UserName*. If *Inet$UserName* is not set, or if it is set to the null string, *getvarusername* returns a NULL pointer (0).

The returned name is null-terminated.

Return value

If the call fails, then a NULL pointer (0) is returned.

Bugs

Host names are limited to MAXHOSTNAMELEN (from *"sys/param.h"*) characters, currently 64.

The return value points to static data whose content is overwritten by each call.

See also

getlogin (page 5a-397)

Exported by

Unixlib

GETWD

Name

getwd – get current working directory pathname

Synopsis

```
char *getwd(pathname)
char *pathname;
```

Description

Getwd copies the pathname of the current working directory to *pathname* and returns a pointer to the result.

Exported by

Unixlib

HERROR

Name

herror – obsolete call

Synopsis

—

Description

Herror is now obsolete, and you must not use it in your code. It is exported from Unixlib only to ensure backwards compatibility.

Exported by

Unixlib

INET

Name

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

Synopsis

```
#include "sys/types.h"

u_long inet_addr(cp)
char *cp;

u_long inet_network(cp)
char *cp;

#include "netinet/in.h"

char *inet_ntoa(in)
struct in_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

int inet_lnaof(in)
struct in_addr in;

int inet_netof(in)
struct in_addr in;
```

Description

The routines *inet_addr* and *inet_network* each interpret character strings representing numbers expressed in the Internet standard '.' notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine *inet_ntoa* takes an Internet address and returns an ASCII string representing the address in '.' notation. The routine *inet_makeaddr* takes an Internet network number and a local network address and constructs an Internet address from it. The routines *inet_netof* and *inet_lnaof* break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet address are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

Internet addresses

Values specified using the '.' notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as '128.net.host'.

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as 'net.host'.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as 'parts' in a '.' notation may be decimal, octal, or hexadecimal, as specified in the C language (ie a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

Return value

The value –1 is returned by *inet_addr* and *inet_network* for malformed requests.

Bugs

The string returned by *inet_ntoa* resides in a static memory area.

See also

gethostbyname (page 5a-394), getnetent (page 5a-398)

Exported by

Inetlib

_INET_ERROR

Name

_inet_error – global error variable

Synopsis

```
#include "kernel.h"
_kernel_oserror _inet_error
```

Description

The global error structure *_inet_error* is used exclusively by the Socklib library. It contains the most recent error block returned from a call into the Internet module, and is set by the function *makecall()*.

Exported by

Socklib

IOCTL

Name

ioctl – control device

Synopsis

```
#include "sys/ioctl.h"

int ioctl(d, request, argp)
int d;
int request;
void *argp;
```

Description

Ioctl is a synonym for *socketioctl*; see page 5a-447. The call is provided mainly so that you do not need to rename *ioctl* calls in code that you are porting.

See also

socketioctl (page 5a-447)

Exported by

Unixlib

KILLFILE

Name

killfile – remove directory entry

Synopsis

```
void killfile(path)
char *path;
```

Description

Killfile removes the entry for the object *path* from its directory. The call fails if the object is locked against deletion, or if it is a directory which is not empty, or if it is already open.

No value is returned.

This call is now deprecated, and we recommend you instead use *unlink* in your code. *Killfile* is exported from Unixlib only to ensure backwards compatibility.

See also

unlink (page 5a-455)

Exported by

Unixlib

LISTEN

Name

listen – listen for connections on a socket

Synopsis

```
int listen(s, backlog)
int s, backlog;
```

Description

To accept connections, a socket is first created with *socket*, a willingness to accept incoming connections and a queue limit for incoming connections are specified with *listen*, and then the connections are accepted with *accept*. The *listen* call applies only to sockets of type SOCK_STREAM.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

Return value

A 0 return value indicates success; –1 indicates an error.

Errors

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[EOPNOTSUPP]	The socket is not of a type that supports the operation *listen*.

Bugs

The *backlog* is currently limited (silently) to 5 and negative numbers are replaced by 0.

It is not the queue length – this is currently defined by:

(*backlog* × 3)/2 + 1

See also

accept (page 5a-371), connect (page 5a-382), socket (page 5a-443)

Exported by

Socklib

LSEEK

Name

lseek – move read/write pointer

Synopsis

```
long lseek(d, offset, whence)
int d;
long offset;
int whence;
```

Description

Lseek sets the file pointer of *d*. Since you cannot seek on sockets, the Unixlib version of *lseek* always fails and the file pointer remains unchanged.

Return value

A value of –1 is always returned.

Errors

Errno is always set to indicate the error.

[ESPIPE]	*d* is associated with a pipe or a socket.

Exported by

Unixlib

_MAKECALL

Name

_makecall – wrapper for SWI calls

Synopsis

```
#include kernel.h

int _makecall(swinum, in, out)
int swinum
_kernel_swi_regs *in, *out
```

Description

Makecall provides a wrapper for calling SWIs, and is used by Socklib for all Socket_… SWIs calls it makes. The first thing *makecall* does is to issue the SWI. Its subsequent action depends on whether or not the SWI returns an error:

- If the SWI does not return an error, the global error variable *errno* is set to zero, and the return value of *makecall* is the value that was in R0 on exit from the SWI.
- If the SWI returns an error, *makecall* copies the returned error block into the global error structure *_inet_error*. It then sets *errno* from the SWI's returned error number, converting standard Internet errors (ie those returned by the SWI in the range &20E00 - &20E7F) to the values used in C by subtracting &20E00. If – after that – the value of *errno* is still greater than EREMOTE, *makecall* then sets *errno* to ESRCH. Finally, *makecall* returns a value of –1.

Exported by

Socklib

NAMISIPADR

Name

namisipadr – get network host entry

Synopsis

```
#include "netdb.h"

struct hostent *namisipadr(name)
char* name
```

Description

Namisipadr takes an Internet address and returns a pointer to an object describing an Internet host. The Internet address is a character string representing numbers expressed in the Internet standard '.' notation; for more details see *Internet addresses* on page 5a-420**.** The information is returned in the following structure:

```
struct hostent {
        char    *h_name;
        char    **h_aliases;
        int     h_addrtype;
        int     h_length;
        char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

The members of this structure are:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A zero terminated array of network addresses for the host. Host addresses are returned in network byte order.
h_addr	The first address in h_addr_list.

Return value

Error return status from *namisipadr* is indicated by return of a null pointer.

Bugs

All information is contained in a static area so it must be copied if it is to be saved.

See also

inet_addr (page 5a-419), gethostbyname (page 5a-394)

Exported by

Inetlib

OSREADC

Name

osreadc – reads a character from the current input stream

Synopsis

```
int osreadc()
```

Description

Osreadc is a veneer to OS_ReadC (page 1-880), which reads a character from the current input stream.

Return value

Osreadc returns the ASCII code of the key pressed, or EOF if Escape was pressed.

See also

xgets (page 5a-458)

Exported by

Unixlib

_PWBUF

Name

_pwbuf – symbol for internal use only

Synopsis

—

Description

The *_pwbuf* symbol is exported for internal use only. You must not use it in your own code.

Exported by

Unixlib

READ

Name

read, readv – read input

Synopsis

```
int read(d, buf, nbytes)
int d;
char *buf;
int nbytes;

#include "sys/types.h"
#include "sys/uio.h"

int readv(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;
```

Description

Read is a synonym for *socketread*, and *readv* a synonym for *socketreadv*; see page 5a-448. These calls are provided mainly so that you do not need to rename *read* and *readv* calls in code that you are porting.

See also

socketread and socketreadv (page 5a-448)

Exported by

Unixlib

READDIR

Name

readdir – read a directory

Synopsis

```
int readdir(path, buf, len, name, offset)
char *path, *buf;
int len, name, offset;
```

Description

Readdir reads an entry from the directory *path* which match the wildcard name *name*; it is returned in the buffer *buf* (which is of length *len*). The *offset* gives the directory entry from which to start searching; it should be zero to start searching from the start of the directory.

If *path* (which may contain wildcards) is a null string, then the currently-selected directory is read. If *name* is a null string then '*' is used, and all files will be matched.

Return value

This call returns the offset from which you should continue searching to read more entries; or –1 if no entry was read, or there are no more entries after the one read by this call.

Bugs

This implementation of *readdir* is **not** a direct replacement or emulation of the Unix *readdir* function. You should bear this especially in mind if you are porting software.

Exported by

Unixlib

RECV

Name

recv, recvfrom, recvmsg – receive a message from a socket

Synopsis

```
int recv(s, buf, len, flags)
int s;
char *buf;
int len, flags;

#include "sys/socket.h"
#include "sys/types.h"

int recvfrom(s, buf, len, flags, from, fromlen)
int s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

#include "sys/uio.h"

int recvmsg(s, msg, flags)
int s;
struct msghdr *msg;
int flags;
```

Description

Recv, *recvfrom*, and *recvmsg* are used to receive messages from a socket.

The *recv* call is normally used only on a *connected* socket, while *recvfrom* and *recvmsg* may be used to receive data on a socket whether it is in a connected state or not.

If *from* is non-zero, the source address of the message is filled in. *Fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see *socket* on page 5a-443).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see *socketioctl* on page 5a-447) in which case a *cc* of –1 is returned with the external variable errno set to EWOULDBLOCK.

The *select* call may be used to determine when more data arrives.

The *flags* argument to a recv call is formed by **or**'ing one or more of the values,

```
#define         MSG_OOB         0x1     /* process out-of-band data */
#define         MSG_PEEK        0x2     /* peek at incoming message */
```

The *recvmsg* call uses a *msghdr* structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in *"sys/socket.h"*:

```
struct msghdr {
        caddr_t         msg_name;       /* optional address */
        int             msg_namelen;    /* size of address */
        struct iovec    *msg_iov;       /* scatter/gather array */
        int             msg_iovlen;     /* # elements in msg_iov */
        caddr_t         msg_accrights;  /* access rights sent/received */
        int             msg_accrightslen;
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter gather locations, as described in *socketread* on page 5a-448. A buffer to receive any access rights sent along with the message is specified in *msg_accrights*, which has length *msg_accrightslen*. Access rights are currently limited to integer values. If access rights are not being transferred, the *msg_accrights* field should be set to NULL.

Return value

These calls return the number of bytes received, or –1 if an error occurred.

Errors

The calls fail if:

[EBADF]	The argument *s* is an invalid descriptor.
[EWOULDBLOCK]	The socket is marked non-blocking and the receive operation would block.
[EFAULT]	The data was specified to be received into an invalid address.

See also

socketread (page 5a-448), send (page 5a-440), select (page 5a-438), getsockopt (page 5a-410), socket (page 5a-443)

Exported by

Socklib

RRESVPORT

Name

rresvport – routine for returning a stream to a remote command

Synopsis

```
int rresvport(port);
int *port;
```

Description

Rresvport is a routine which returns a descriptor to a socket with an address in the privileged port space bound to it. Privileged Internet ports are those in the range 0 to 1023.

Return value

Rresvport returns a valid, bound socket descriptor on success. It returns –1 on error with the global value *errno* set according to the reason for failure (see *ERRNO* on page 5a-384). The error code EAGAIN is overloaded to mean ‘All network ports in use’.

Exported by

Inetlib

SELECT

Name

select – synchronous socket I/O multiplexing

Synopsis

```
#include "sys/types.h"
#include "sys/time.h"

int select (nfds,readfds,writefds,exceptfds,timeout)
int nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

Description

Select examines the socket descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; ie the descriptors from 0 through *nfds*–1 in the descriptor sets are examined. On return, *select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in *nfound*.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: *FD_ZERO(&fdset)* initializes a descriptor set *fdset* to the null set. *FD_SET(fd, &fdset)* includes a particular descriptor *fd* in *fdset*. *FD_CLR(fd, &fdset)* removes *fd* from *fdset*. *FD_ISSET(fd, &fdset)* is nonzero if *fd* is a member of *fdset*, zero otherwise. The behaviour of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select blocks indefinitely. To affect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as zero pointers if no descriptors are of interest.

Return value

Select returns the number of ready descriptors that are contained in the descriptor sets, or –1 if an error occurred. If the time limit expires then *select* returns 0. If *select* returns with an error, the descriptor sets will be unmodified.

Errors

An error return from *select* indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.

See also

accept (page 5a-371), connect (page 5a-382), socketread (page 5a-448), socketwrite (page 5a-452

Exported by

Socklib

SEND

Name

send, sendto, sendmsg – send a message from a socket

Synopsis

```
int send(s, msg, len, flags)
int s;
char *msg;
int len, flags;

#include "sys/socket.h"
#include "sys/types.h"

int sendto(s, msg, len, flags, to, tolen)
int s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;

#include "sys/uio.h"

int sendmsg(s, msg, flags)
int s;
struct msghdr *msg;
int flags;
```

Description

Send, *sendto*, and *sendmsg* are used to transmit a message to another socket. *Send* may be used only when the socket is in a *connected* state, while *sendto* and *sendmsg* may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of –1 indicate some locally detected errors.

If no messages space is available at the socket to hold the message to be transmitted, then *send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The select call (page 5a-438) may be used to determine when it is possible to send more data.

The *flags* parameter may be set to MSG_OOB (otherwise 0) to send ‘out-of-band’ data on sockets that support this notion (eg SOCK_STREAM); the underlying protocol must also support ‘out-of-band’ data.

See *recv* for a description of the *msghdr* structure.

Return value

The call returns the number of characters sent, or –1 if an error occurred.

Errors

The call fails if:

[EBADF]	An invalid descriptor was specified.
[EFAULT]	An invalid address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the requested operation would block.
[ENOBUFS]	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
[ENOBUFS]	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

See also

recv (page 5a-434), select (page 5a-438), getsockopt (page 5a-410), socket (page 5a-443), socketwrite (page 5a-452)

Exported by

Socklib

SHUTDOWN

Name

shutdown – shut down part of a full-duplex connection

Synopsis

```
int shutdown(s, how)
int s, how;
```

Description

The *shutdown* call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

Return value

A 0 return value indicates success; –1 indicates an error.

Errors

The call fails if:

[EBADF]	*s* is not a valid descriptor.
[ENOTCONN]	The specified socket is not connected.
[ENOTSOCK]	*s* is a file, not a socket.

See also

connect (page 5a-382), socket (page 5a-443)

Exported by

Socklib

SOCKET

Name

socket – create an endpoint for communication

Synopsis

```
int socket(domain, type, protocol)
int domain, type, protocol;
```

Description

Socket creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. The currently understood format under RISC OS is

PF_INET (Internet protocols).

The socket has the indicated *type*, which specifies the semantics of communication. Currently defined types under RISC OS are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network protocols and interfaces.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the ‘communication domain’ in which communication is to take place; see *Protocols* on page 5a-361.

Sockets of type SOCK_STREAM are full-duplex byte streams. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect* call. Once connected, data may be transferred

using some variant of the *send* and *recv* calls. When a session has been completed a *socketclose* may be performed. Out-of-band data may also be transmitted as described in *send* and received as described in *recv.*

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with –1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets 'warm' by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (eg 5 minutes). An *Internet_Event/Socket_Broken_Event* event occurs if a program sends on a broken stream, provided the program has enabled the event (see *The Internet event* on page 5a-345)

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send* calls. Datagrams are generally received with *recvfrom*, which returns the next datagram with its return address.

The operation of sockets is controlled by socket level *options*. These options are defined in the file *"sys/socket.h"*. *Setsockopt* and *getsockopt* are used to set and get options, respectively.

Return value

A –1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

Errors

The *socket* call fails if:

[EPROTONOSUPPORT]	The protocol type or the specified protocol is not supported within this domain.
[EMFILE]	The socket descriptor table is full.
[EACCES]	Permission to create a socket of the specified type and/or protocol is denied.
[ENOBUFS]	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

See also

accept (page 5a-371), bind (page 5a-375), connect (page 5a-382), getsockname (page 5a-409), getsockopt (page 5a-410), socketioctl (page 5a-447), listen (page 5a-424), socketread (page 5a-448), recv (page 5a-434), select (page 5a-438), send (page 5a-440), shutdown (page 5a-442), socketwrite (page 5a-452)

Exported by

Socklib

SOCKETCLOSE

Name

socketclose – close an open socket

Synopsis

```
int socketclose(s)
int s;
```

Description

Socketclose closes an open socket, and releases any resources, including queued data, associated with it.

Because RISC OS uses a global descriptor table, you can close another program's socket. You must therefore take care that the socket descriptor passed belongs to your program.

If an application terminates under RISC OS without closing an open socket, then that socket will remain open indefinitely, needlessly consuming resources. You therefore **must** ensure your applications close all sockets before quitting.

Return value

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and the global integer variable *errno* is set to indicate the error.

Errors

The call will fail if:

[EBADF]	*s* is not a valid descriptor.

See also

close (page 5a-381), accept (page 5a-371), socket (page 5a-443)

Exported by

Socklib

SOCKETIOCTL

Name

socketioctl – control an open socket

Synopsis

```
int socketioctl(s, request, argp)
int s;
unsigned long request;
void *argp;
```

Description

Socketioctl is used to alter the operating characteristics of an open socket, *s*. The parameter *request* specifies the socketioctl command, and has encoded within it both the size of the argument pointed to by *argp*, and whether the argument is an ‘in’ parameter or an ‘out’ parameter. Macros and defines used in specifying a socketioctl *request* are located in the header file *"sys/ioctl.h"*.

Return value

If an error has occurred, a value of –1 is returned and errno is set to indicate the error.

Errors

The call will fail if:

[EBADF]	*s* is not a valid descriptor.
[ENOTTY]	The specified request does not apply to the kind of object that the descriptor *d* references.
[EINVAL]	*Request* or *argp* is not valid.

Exported by

Socklib

SOCKETREAD

Name

socketread, socketreadv – read input

Synopsis

```
int socketread(d, buf, nbytes)
int d;
char *buf;
int nbytes;

#include "sys/types.h"
#include "sys/uio.h"

int socketreadv(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;
```

Description

Socketread attempts to read *nbytes* of data from the object referenced by the descriptor *d* into the buffer pointed to by *buf*. *Socketreadv* performs the same action, but scatters the input data into the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], …, iov[iovcnt–1].

For *socketreadv*, the *iovec* structure is defined as

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. *Socketreadv* will always fill an area completely before proceeding to the next.

Upon successful completion, *socketread* and *socketreadv* return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file, but in no other case.

If the returned value is 0, then end-of-file has been reached.

Return value

If successful, the number of bytes actually read is returned. Otherwise, a –1 is returned and the global variable *errno* is set to indicate the error.

Errors

Socketread and *socketreadv* will fail if one or more of the following are true:

[EBADF]	*D* is not a valid socket descriptor open for reading.
[EFAULT]	*Buf* points outside the allocated address space.
[EIO]	An I/O error occurred while reading from the socket.
[EINTR]	A read from a slow device was interrupted before any data arrived.
[EINVAL]	The pointer associated with *d* was negative.
[EWOULDBLOCK]	The socket was marked for non-blocking I/O, and no data were ready to be read.

In addition, *socketreadv* may return one of the following errors:

[EINVAL]	*Iovcnt* was less than or equal to 0, or greater than 16.
[EINVAL]	One of the *iov_len* values in the *iov* array was negative.
[EINVAL]	The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.
[EFAULT]	Part of the *iov* points outside the program's allocated address space.

See also

select (page 5a-438), socket (page 5a-443)

Exported by

Socklib

SOCKETSTAT

Name

socketstat – get socket status

Synopsis

```
#include "sys/types.h"
#include "sys/stat.h"

int socketstat(s, buf)
int s;
struct stat *buf;
```

Description

Socketstat obtains information about the socket descriptor *s*.

Buf is a pointer to a *stat* structure into which information is placed concerning the socket. The contents of the structure pointed to by *buf* are:

```
struct stat {
        dev_t   st_dev;         /* device inode resides on */
        ino_t   st_ino;         /* this inode's number */
        u_short st_mode;        /* protection */
        short   st_nlink;       /* number or hard links to the file */
        uid_t   st_uid;         /* user-id of owner */
        gid_t   st_gid;         /* group-id of owner */
        dev_t   st_rdev;        /* device type, for inode that is device */
        off_t   st_size;        /* total size of file */
        time_t  st_atime;       /* file last access time */
        int     st_spare1;
        time_t  st_mtime;       /* file last modify time */
        int     st_spare2;
        time_t  st_ctime;       /* file last status change time */
        int     st_spare3;
        long    st_blksize;     /* optimal blocksize for file system i/o */
        long    st_blocks;      /* actual number of blocks allocated */
        long    st_spare4[2];
};
```

The *st_blksize* field may be either updated or preserved, depending on the socket's protocol. All other fields have little or no meaning for sockets, and are preserved.

Return value

Upon successful completion a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Errors

Socketstat will fail if one or more of the following are true:

[EBADF] *s* is not a valid descriptor.

Exported by

Socklib

SOCKETWRITE

Name

socketwrite, socketwritev – write output

Synopsis

```
int socketwrite(d, buf, nbytes)
int d;
char *buf;
int nbytes;

#include "sys/types.h"
#include "sys/uio.h"

int socketwritev(d, iov, iovcnt)
int d;
struct iovec *iov;
int iovcnt;
```

Description

Socketwrite attempts to write *nbytes* of data to the object referenced by the descriptor *d* from the buffer pointed to by *buf*. Socket*writev* performs the same action, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], …, iov[iovcnt–1].

For *socketwritev*, the *iovec* structure is defined as

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. *Socketwritev* will always write a complete area before proceeding to the next.

Sockets are subject to flow control, so *socketwrite* and *socketwritev* may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

Return value

Upon successful completion the number of bytes actually written is returned. Otherwise a –1 is returned and the global variable *errno* is set to indicate the error.

Errors

Socketwrite and *socketwritev* will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF]	*D* is not a valid descriptor open for writing.
[EPIPE]	An attempt is made to write to a socket of type SOCK_STREAM that is not connected to a peer socket.
[EFAULT]	Part of *iov* or data to be written to the socket points outside the program's allocated address space.
[EINVAL]	The pointer associated with *d* was negative.
[EIO]	An I/O error occurred while reading from or writing to the socket.
[EWOULDBLOCK]	The socket was marked for non-blocking I/O, and no data could be written immediately.

In addition, *socketwritev* may return one of the following errors:

[EINVAL]	*Iovcnt* was less than or equal to 0, or greater than 16.
[EINVAL]	One of the *iov_len* values in the *iov* array was negative.
[EINVAL]	The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.

See also

select (page 5a-438)

Exported by

Socklib

STRING

Name

strcasecmp, strncasecmp, index, rindex – string operations

Synopsis

```
int strcasecmp(s1, s2)
char *s1, *s2;

int strncasecmp(s1, s2, n)
char *s1, *s2;
int n

char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s, c;
```

Description

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

Strcasecmp compares its arguments and returns an integer of 1 or 0, according as *s1* is lexicographically not equal to, or equal to *s2*. *Strncasecmp* makes the same comparison but looks at at most *n* characters.

Index (*rindex*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

Exported by

Unixlib

UNLINK

Name

unlink – remove directory entry

Synopsis

```
int unlink(path)
char *path;
```

Description

Unlink removes the entry for the object *path* from its directory. The call fails if the object is locked against deletion, or if it is a directory which is not empty, or if it is already open.

Return value

A value of 0 is always returned.

Exported by

Unixlib

_VARNAMEBUF

Name

_varnamebuf – call for internal use only

Synopsis

—

Description

The *_varnamebuf* symbol is exported for internal use only. You must not use it in your own code.

Exported by

Unixlib

WRITE

Name

write, writev – write output

Synopsis

```
int write(d, buf, nbytes)
int d;
char *buf;
int nbytes;

#include "sys/types.h"
#include "sys/uio.h"

int writev(d, iov, iovcnt)
int d;
struct iovec *iov;
int iovcnt;
```

Description

Write is a synonym for *socketwrite*, and *writev* a synonym for *socketwritev*; see page 5a-452. These calls are provided mainly so that you do not need to rename *write* and *writev* calls in code that you are porting.

See also

socketwrite and socketwritev (page 5a-452)

Exported by

Unixlib

XGETS

Name

xgets – get a string from a stream

Synopsis

```
char *xgets(s)
char *s;
```

Description

Xgets reads a string into *s* from the current input stream. The string is terminated by a return character, which is replaced in *s* by a linefeed character; or by EOF. The last character read into *s* is followed by a null character. *Xgets* returns its argument.

Exported by

Unixlib

XPERROR

Name

xperror, sys_errlist, sys_nerr – system error messages

Synopsis

```
void xperror(s)
const char *s;

char *sys_errlist[];

int sys_nerr;
```

Description

Xperror produces a short error message on the current output stream describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the newline. *Sys_nerr* is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table. Indeed, *xperror* makes such a check, and outputs a default message string if *errno* exceeds *sys_nerr.*

See also

errno (page 5a-384)

Exported by

Unixlib

XPUTCHAR

Name

xputchar – put character or word on a stream

Synopsis

```
char xputchar(c)
char c;
```

Description

Xputchar appends the character *c* to the current output stream. It returns the character written.

Exported by

Unixlib

Service calls

This section documents the service calls that are used to communicate between network device drivers and the rest of RISC OS. Unfortunately, you need to know and understand the Driver Control Interface specification before you can follow all the details of these descriptions, and that is beyond the scope of this manual to document. See *The software in detail* on page 5a-286.

Driver information blocks

A device driver identifies each logical interface it controls by a *driver information block.* These are used by a number of the service calls that follow, and have the following structure:

Offset	Value
0	The base of the device driver's allocated SWI chunk
4	Pointer to the device driver's unique short name, null terminated (eg 'en', 'ppp')
8	The unit number
12	Pointer to 6 bytes giving the hardware address of the interface
16	Pointer to the device driver's title, null terminated (eg 'Ether3')
20	Pointer to a string describing the physical location of the interface, null terminated (eg 'Network Expansion Slot', 'Expansion Slot 0, port #1')
24	Bitfield specifying physical slot: bits 0 - 7 slot number: 0 - 7 ⇒ physical expansion card slot 8 ⇒ network expansion card slot 128 ⇒ parallel port 129 ⇒ Serial port (eg PPP) 130 ⇒ Econet socket 131 ⇒ PCMCIA card bits 8 - 15 reserved for device driver's use: may be used where one card provides multiple independent interfaces bits 16 - 20 physical PCMCIA slot, used when slot number is 131 bits 21 -31 reserved – must be zero

Offset	Value
28	Bitfield giving characteristics of device driver; meaning when set is:
	bit 0 — multicast reception is supported
	bit 1 — promiscuous reception is supported
	bit 2 — interface receives its own transmitted packets
	bit 3 — station number required
	bit 4 — interface can receive erroneous packets
	bit 5 — interface has a hardware address
	bit 6 — driver can alter interface's hardware address
	bit 7 — interface is a point to point link
	bit 8 — driver supplies standard statistics
	bit 9 — driver supplies extended statistics
	bit 10 — interface is virtual (ie a software emulation of hardware); refer to the Driver Control Interface specification
	bits 11 - 31 — reserved – must be zero

Each driver information block must be held as static data. In this way, protocol modules can identify an interface simply by comparing the addresses of driver information blocks, rather than by laboriously comparing fields within the block.

This does mean that any use of the *RMTidy command will kill any network stack on the machine. In practice this is unlikely to be a great problem, since this command has long been deprecated.

The service call Service_DCIDriverStatus (page 5a-464) provides a mechanism for a device driver module that is re-initialised (via *RMReInit) to pass the new address of its driver information block to a protocol module.

Service_EnumerateNetworkDrivers (Service Call &9B)

Issued to obtain a linked list of all active network device drivers

On entry

R0 = pointer to head of linked list of network device drivers
R1 = &9B (reason code)

On exit

R0 = pointer to new head of linked list of network device drivers
R1 preserved to pass on call

Use

This service call is issued to obtain a linked list of all active network device driver modules in the system. When the service call is first issued, R0 is a null pointer. When a device driver receives this call, it should chain an entry to the **head** of the linked list for each logical interface it controls. Each entry is as follows:

Offset	Value
0	Pointer to the next entry in the linked list; the last entry is null
4	Pointer to the driver information block for this entry

These entries are transient objects: the device drivers should allocate the space for them from RMA using OS_Module 6 (page 1-237), and the program that issued the service call should free them back into RMA using OS_Module 7 (page 1-238) after the call returns. However, the driver information blocks referenced by each entry in the linked list must be static data; see *Driver information blocks* on page 5a-461.

You must not claim this service call.

Service_DCIDriverStatus (Service Call &9D)

Issued by a network device driver module once initialised, and when finalising

On entry

R0 = pointer to driver information block for starting/terminating driver
R1 = &9D (reason code)
R2 = status (0 ⇒ starting, 1 ⇒ terminating)
R3 = DCI version supported × 100 (eg 403 for version 4.03)

On exit

R0 - R3 preserved

Use

This service call is issued by a network device driver module once it has initialised and is ready to accept SWI calls (R2 = 0), and when it is finalising and can no longer accept SWI calls (R2 = 1). If the device driver controls multiple logical interfaces, it issues this service call once for each interface.

When a protocol module receives this service call from a driver that is starting up (ie R2 = 0), it should add the driver to its list of device drivers. When a protocol module receives the call from a driver that is terminating (ie R2 = 1), it should scan its list of device drivers for a driver information block matching the one addressed by R0, and remove the corresponding driver from the list.

You may instead decide to keep a terminating driver on the list, but to mark it as inactive in case it later restarts. If you do this, you must not match driver information blocks by comparing their addresses, as when a driver restarts the block may well be in a different location. You instead have to match fields within the driver information block: the short name and unit number (at offsets 4 and 8 respectively) together uniquely identify a driver information block, and so a match of them is sufficient.

You must not claim this service call.

Service_DCIFrameTypeFree (Service Call &9E)

This Service Call requires you to use the Driver Control Interface, and so is beyond the scope of this manual to document. See *The software in detail* on page 5a-286.

Service_DCIProtocolStatus (Service Call &9F)

Issued by a protocol module once it has initialised, and when it is finalising

On entry

R0 = protocol module's private word pointer
R1 = &9F (reason code)
R2 = status (0 ⇒ starting, 1 ⇒ terminating)
R3 = DCI version supported × 100 (eg 403 for version 4.03)
R4 = pointer to protocol module's title string

On exit

R0 - R4 preserved

Use

This service call is issued by a protocol module once it has initialised and is ready to accept SWI calls (R2 = 0), and when it is finalising and can no longer accept SWI calls (R2 = 1).

The title string pointed to by R4 should be identical to the title string in the protocol module's header. This string is not used anywhere else in the DCI. (It is intended for use by modules that rely on the protocol module, but which do not communicate with it via the DCI; these modules need to have the name of significant protocol modules built into them.)

When a terminating protocol module issues this service call, it must continue handling receive events for all frame types it has not explicitly relinquished, until the service call returns. Once the call has returned, device drivers should have deleted **all** references to the protocol module which issued the service call.

This supersedes the unnamed service call &41200, which was never part of any formal DCI specification, but which used to be issued during finalisation of the Internet module.

You must not claim this service call.

Service_InternetStatus (Service Call &B0)

Issued by the Internet module when an interface address has been changed

On entry

R0 = 0 (subreason code)
R1 = &B0 (reason code)

On exit

R0, R1 preserved

Use

This service call is issued by the Internet module upon successful completion of an SIOCSIFADDR *socketioctl()* call; ie when an interface address has been changed.

You must not claim this service call.

SWI calls

There is a direct SWI equivalent to each call available in Socklib. In fact when you make a call to Socklib, all that happens is that the parameters you pass are loaded into the ARM processor's registers, and the relevant SWI is issued; any returned RISC OS error block is converted to a C error number.

Calling the SWIs

You may wish to issue the SWIs yourself – say if you're programming in BASIC. The table below shows you how each Socket_… SWI corresponds to the Socklib calls documented elsewhere in this chapter, giving the name and number of the SWI, the equivalent Socklib call, and the page on which it is documented:

SWI name	SWI no	Socklib call	Page
Socket_Accept	&41203	accept	5a-371
Socket_Bind	&41201	bind	5a-375
Socket_Close	&41210	socketclose	5a-446
Socket_Connect	&41204	connect	5a-382
Socket_Creat	&41200	socket	5a-443
Socket_Getpeername	&4120E	getpeername	5a-401
Socket_Getsockname	&4120F	getsockname	5a-409
Socket_Getsockopt	&4120D	getsockopt	5a-410
Socket_Gettsize	&41218	getstablesize	5a-413
Socket_Ioctl	&41212	socketioctl	5a-447
Socket_Listen	&41202	listen	5a-424
Socket_Read	&41213	socketread	5a-448
Socket_Readv	&41216	socketreadv	5a-448
Socket_Recv	&41205	recv	5a-434
Socket_Recvfrom	&41206	recvfrom	5a-434
Socket_Recvmsg	&41207	recvmsg	5a-434
Socket_Select	&41211	select	5a-438
Socket_Send	&41208	send	5a-440
Socket_Sendmsg	&4120A	sendmsg	5a-440
Socket_Sendto	&41209	sendto	5a-440
Socket_Sendtosm	&41219	*Reserved for internal use*	—
Socket_Setsockopt	&4120C	setsockopt	5a-410
Socket_Shutdown	&4120B	shutdown	5a-442

SWI name	SWI no	Socklib call	Page
Socket_Stat	&41215	socketstat	5a-450
Socket_Write	&41214	socketwrite	5a-452
Socket_Writev	&41217	socketwritev	5a-452

Passing parameters

When calling a Socket_… SWI, you pass the parameters from the corresponding Socklib call in registers R0 upwards: the first parameter in R0, the second in R1, and so on.

Say you wish to call Socket_Accept. The equivalent call is *accept*:

```
int accept(s, addr, addrlen)
```

Therefore when calling the SWI, you would pass the parameter *s* in R0, *addr* in R1, and *addrlen* in R2.

Any returned value is passed back in R0: since the *accept* call returns an int, this will be returned in R0 for Socket_Accept.

Errors

Errors from Socket_… SWI calls are indicated in the standard way used by RISC OS:

- If the V (overflow) flag is clear on return from a SWI, then no error occurred and the desired action was performed.
- If the V flag is set, then an error occurred, and R0 points to an error block, the first word of which contains an error number in the Internet module's reserved range of error numbers (&20E00 - &20EFF). The rest of the error block consists of a null-terminated error message.

The lower half of the error numbers (ie &20E00 - &20E7F) are used for standard Internet errors. These are &20E00 greater than the corresponding Unix error number placed in the `errno` variable after a Socklib call. For example, EINVAL is returned from Socket_… SWI calls as &20E16, but is returned from Socklib calls as &16 – as defined in the C header files.

The upper half of the error numbers (ie &20E80 - &20EFF) are used for errors that are specific to DCI4 and later.

For a full description of how Socklib library calls deal with errors returned from Socket_… SWIs, see the description of *_makecall* on page 5a-427.

* Commands

*InetChecksum

Enables or disables various protocol checksums

Syntax

```
*InetChecksum [i|u|t On|Off]
```

Parameters

`i`	set IP checksum usage
`u`	set UDP checksum usage
`t`	set TCP checksum usage
`On`	enable checksums
`Off`	disable checksums

Use

*InetChecksum enables or disables various protocol checksums, or reports the current state of the checksums if you pass no parameters. You should not alter these values unless you know what you are doing.

Example

```
*InetChecksum
IP checksums are off, UDP checksums are off, TCP checksums are on

*InetChecksum u On
```

Related commands

None

*InetGateway

May be used to enable or to disable IP layer packet forwarding

Syntax

```
*InetGateway [On|Off]
```

Parameters

`On`	enable IP layer packet forwarding
`Off`	disable IP layer packet forwarding

Use

*InetGateway may be used to enable or to disable IP layer packet forwarding (ie gateway operation) if multiple network interfaces are present. With no parameters, the current forwarding status is reported.

The default state is off.

Example

```
*InetGateway
Packet forwarding not in operation

*InetGateway On
```

Related commands

None

*InetInfo

Displays Internet module internal statistics

Syntax

```
*InetInfo [r] [i] [p]
```

Parameters

r	give details of internal resource usage (the default)
i	give details of interfaces fitted
p	give details of protocols

Use

*InetInfo displays information and statistics about the current state of the Internet module, which forms a part of the AUN software. Most of the information displayed is runic in nature. It is presented mainly as an aid to trouble-shooting, should you require it.

Example

```
*InetInfo r

Resource usage:

Sockets
    Active 10

Packet forwarding not in operation
```

Related commands

None

124 Acorn Access

Introduction and Overview

Acorn Access is Acorn's entry level product for AUN networking. It provides peer to peer networking using TCP/IP protocols, allowing sharing of resources such as discs and printers.

From RISC OS 3.6 onwards, Access is supplied as a part of the operating system.

Access components

There are three main modules that make up Access: Freeway, ShareFS and RemotePrinterSupport.

Freeway

Freeway provides the protocols used by Access so it knows what shared resources are available and can display windows showing them.

- The interfaces used by Freeway are for internal use only; you must not use them in your own code.

ShareFS

ShareFS is a filing system that is used to share resources, both granting other hosts access to your machine, and vice versa.

- ShareFS provides * Commands that you can use to share your own filing systems with other Access users. The use of these is described in ** Commands* on page 5a-475.
- It also provides SWIs that you can use to add a **Share** option to a Filer's icon bar menu; see the section *Writing Filers so they integrate with Access* on page 5a-474.

RemotePrinterSupport

RemotePrinterSupport provides the support needed to share printers over an Access network.

- The interfaces used by RemotePrinterSupport are for internal use only; you must not use them in your own code.

Writing filing systems so they integrate with Access

The Access * Commands call standard entry points to filing systems when making them shared. You do not need to take any special steps to make a filing system work with Access; any filing system will work, provided it conforms to the specifications in the chapters *Writing a filing system* on page 2-531 and page 5a-261; and (where applicable) *Writing a FileCore module* on page 2-597 and page 5a-265.

Writing Filers so they integrate with Access

For a Filer to integrate with Access, it needs to provide a **Share** menu option, and take appropriate action when the option is chosen. This is done using a SWI interface to ShareFS.

These SWIs are subject to change as the range of Acorn networking products is expanded and updated, so we don’t document them here. Should you wish to write a Filer to integrate with Access, you should contact Acorn Customer Services.

* Commands

*Desktop_ShareFSFiler

Command to start up ShareFS Filer

Syntax

```
*Desktop_ShareFSFiler
```

Parameters

None

Use

*Desktop_… commands are used by the Desktop to start up ROM-resident Desktop utilities that appear automatically on the icon bar. However, they are for internal use only, and you should not use them; use *Desktop instead.

See page 3-277 for further details of *Desktop_… commands.

Related commands

*Desktop (page 3-275)

*Dismount

Ensures that it is safe to finish using a remote shared disc

Syntax

`*Dismount :`*`disc_name`*

Parameters

`disc_name`	the name of the remote shared disc

Use

*Dismount ensures that it is safe to finish using a remote shared disc by closing all its files, unsetting all its directories and libraries, and discarding its local caches.
*Dismount is useful before finishing sharing a particular disc. However, the *Shutdown command is usually to be preferred, especially when switching off the computer.

Example

```
*Dismount :Maths
```

Related commands

*Shutdown (page 2-188)

*Free

Displays the total free space remaining on a remote shared disc

Syntax

`*Free :`*`disc_name`*

Parameters

`disc_name` the name of the remote shared disc

Use

*Free displays the total free space remaining on a remote shared disc.

Example

```
*Free :Maths
Bytes free &00504400 =   5260288
Bytes used &02413c00 =  37829632
```

Related commands

None

*FwShow

Displays all currently known Freeway objects

Syntax

```
*FwShow
```

Parameters

None

Use

FwShow displays all currently known Freeway objects. Local objects are prefixed with a ''.

Example

```
*FwShow
No remote nets

Type 2:

Type 5: (Hosts)
   *Name=794148708  Holder=1.97.238.89
    Name=528163826  Holder=1.97.238.93
    Name=873634028  Holder=1.97.238.88

Type 1: (Discs)
    Name=English    Holder=1.97.238.93
    Name=Science    Holder=1.97.238.88
   *Name=Maths      Holder=1.97.238.89
```

Related commands

None

*Share

Makes a local directory available as a shared disc

Syntax

`*Share` *`directory`* [*`disc_name`*] [`-protected`] [`-cdrom`] [`-noicon`]

Parameters

`directory`	a valid pathname specifying a directory
`disc_name`	the name to use for the shared disc
`-protected`	causes the directory to be shared protected, rather than the default of unprotected
`-cdrom`	indicates that the shared directory is on a CD-ROM
`-noicon`	prevents an icon appearing for the shared disc

Use

*Share makes a local directory available as a shared disc. If no name is given for the shared disc, then the name of the directory is used, or – for the root directory – the name of the disc itself.

If the directory is shared unprotected, then remote users have read and write access to all objects beneath it. If the directory is shared protected, then remote users' access to an object beneath it is determined by that object's public access attributes.

Example

```
*Share ADFS::Maths.$ Maths -protected
```

Related commands

*Shares, *UnShare

*ShareFS

Selects the Shared Filing System as the current filing system

Syntax

```
*ShareFS
```

Parameters

None

Use

*ShareFS selects the Shared Filing System as the filing system for subsequent operations. Remember that it is not necessary to switch filing systems if you use the full pathnames of objects. For example, you can refer to ADFS objects (on a local disc) when ShareFS is the current filing system.

Example

```
*ShareFS
```

Related commands

*ADFS, *Net, *RAM, *ResourceFS

*ShareFSIcon

Adds an icon to the icon bar for a remote shared disc

Syntax

`*ShareFSIcon` *`disc_name`*

Parameters

`disc_name`	the name of the remote shared disc

Use

*ShareFSIcon adds an icon to the icon bar for a remote shared disc.

Example

```
*ShareFSIcon Maths
```

Related commands

None

*ShareFSWindow

Changes or reports the size of the ShareFS transmission window

Syntax

```
*ShareFSWindow [size]
```

Parameters

size	the size of the ShareFS transmission window

Use

*ShareFSWindow changes the size of the ShareFS transmission window, or – with no parameter – reports its current size. You should not change the size unless you know what you are doing.

Example

```
*ShareFSWindow
Current ShareFS window size: 2
```

Related commands

None

*Shares

*Shares lists the local directories currently made available as shared discs

Syntax

```
*Shares
```

Parameters

None

Use

*Shares lists the local directories currently made available as shared discs, showing the full *Share command with which it was shared.

Example

```
*Shares
Share ADFS::Maths.$ Maths -protected
```

Related commands

*Share

*UnShare

*UnShare makes a local directory no longer available as a shared disc

Syntax

`*UnShare` *`disc_name`*

Parameters

`disc_name`	the name of the remote shared disc

Use

*UnShare makes a local directory no longer available as a shared disc.

Example

```
*Unshare Maths
```

Related commands

*Share

Part 17 – The desktop

125 The desktop

Introduction and Overview

Major changes were made to the desktop in RISC OS 3.5, many of them to improve its appearance. This introduction outlines those changes; the rest of the chapter details the changes they have introduced to the programmers' interface. Incidentally, some of the changes below are entirely handled by RISC OS, and so have introduced no new interfaces.

Desktop appearance

There have been sprite and template changes to give a 3D appearance to the windows. You should refer to the RISC OS 3 Style Guide for more information in this area.

The desktop now uses a proportional font in the desktop and can tile the window backgrounds with a texture.

The Filer

The Filer was changed so that:

- the directory displays can have a variable column width
- all text uses the current desktop font
- filenames of up to 63 characters can be displayed, rather than 10 as before
- objects are dragged as an icon rather than as an outline
- open directories are differentiated using a new icon.

New error system

The Wimp error messages were changed to be more helpful, consistent and user friendly. Applications can now provide a more suitable wording on Error messages and buttons.

The Pinboard

The Pinboard was changed to support outline fonts. The *Backdrop command was extended so you can remove a backdrop.

DragASprite

DragASprite was changed so that the dragged sprite will by default be dithered (ie semi-transparent). You can then see the area underneath a drag, and hence where you are moving a large sprite.

The Watchdog

Currently, if a program goes into an 'infinite loop' (eg it keeps posting an error box without polling) there is no way to stop it. The Wimp now has a watchdog triggered by a hot-key combination, which can be used to kill such rogue programs.

Terminology

Throughout this chapter, when we refer to the desktop's *system font*, we are referring to text that the Wimp outputs using VDU calls (as in earlier versions), and **not** to text output using the outline System.Fixed font.

Technical Details

The desktop font

In previous versions of RISC OS, the Wimp plotted text in icons using the OS_Write… calls and VDU commands; this uses the *system font*, which is a bit-mapped fixed width font. From RISC OS 3.5 onwards the Wimp can instead call the Font Manager, and use a single proportionally spaced *outline font* for text output.

In this chapter, we shall refer to the current font used by the Wimp as the *desktop font*, whether it be an outline font or the system font.

If painting an outline font generates an error for any reason, then the Wimp does not report an error, but reverts to the system font. This avoids loops where reporting an outline font error generates the same error itself.

The WIMPSymbol font

There are some characters that are present in the system font and used in the desktop, but are not present in most outline fonts. A new font named WIMPSymbol has been created that holds outline versions of the most commonly used such characters. It only has these characters defined:

Code	Character	Source
&80	3	Selwyn, &62
&84	7	Selwyn, &63
&88	⇐	Sydney, &DC
&89	⇒	Sydney, &DE
&8A	⇓	Sydney, &DF
&8B	⇑	Sydney, &DD

If the Wimp is using an outline font, it switches to the WIMPSymbol font when outputting these characters.

Templates

Templates and outline fonts

When designing templates, you should ensure that they work with the system font, with Homerton Medium at 12 point, and (preferably) with Trinity Medium at 12 point.

Where text may change, ensure there is enough space for a ‘worst case’. To help you in this, you may find it useful to know that the widest Homerton character is ‘@’, and the widest alphanumeric character is ‘W’. In Trinity the corresponding characters are ‘…’ (amongst others), and ‘W’.

You should be aware that you can no longer use spaces to align columns (such as those in the Filer’s Full Info output). Instead you must use a separate icon for each column, or use the new SWI Wimp_TextOp 2 (see page 5a-505) within a redraw loop.

2D and 3D templates

There is a CMOS bit that users can set to indicate whether they prefer to use a 2D or 3D desktop (see *CMOS RAM allocation* on page 5a-73).

We do not require your application to provide both 2D and 3D templates. Should you choose to do so, you should select the appropriate set by examining this bit at startup, and whenever there is a mode change.

Sprite area control block pointers

When the Wimp loads a template, it now forces the sprite area control block pointer in any window definitions to 1. This is because some template editors do not set this field correctly, but the Wimp now uses it to search for a tiling sprite (see *Tiled window backdrops* on page 5a-492); an invalid value can have disastrous consequences.

Menus

The Wimp works out menu widths for you, even for outline fonts. Your application need not worry about setting the correct width for a menu entry – except for a writable field, when the supplied width will be used as minimum. Menus will be just wide enough to contain the title, and all of the entries, in the menu.

Shortcuts

In menus, keyboard shortcuts must be displayed right-aligned. Previously this was done by using spaces to align the shortcuts, but this is no longer possible with outline fonts. From RISC OS 3.5 onwards the Wimp automatically finds and right aligns any shortcut at the end of a menu entry, using simple rules.

For the Wimp to recognise a menu entry as having a shortcut, both the following must be true:

- The menu entry must be non-writable.
- It must contain at least one space.

and at least one of the following must also be true:

- The string after the last space must start with no more than one of the patterns in the *Modifiers* list, held in the Wimp's Messages file. In the UK this list is:
  ⇑ ^ ^⇑ ⇑^
- The entry must end with a pattern from the *KeyNames* list, also held in the Wimp's Messages file. This list consists of:

  Esc ESC F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 Print PRINT Break BREAK Pause PAUSE Tab TAB Return RETURN Insert INSERT Home HOME PageUp PAGE UP Delete DELETE Copy COPY End END PageDown PAGE DOWN Enter ENTER Up UP Down DOWN Left LEFT Right RIGHT Select SELECT Menu MENU Adjust ADJUST

If the above conditions are satisfied, the Wimp right aligns everything after the last space.

By holding the lists in the Wimp's messages file, they can be internationalised, so (for example) the modifiers could be a whole word rather than just a symbol. We encourage you to follow the guidelines in the *RISC OS 3 Style Guide* for giving shortcuts.

Icon bar icons that use text

Some applications put icons on the icon bar that have text as well as a sprite. Obviously the width of such icons can change as the desktop font is changed. From RISC OS 3.5 onwards, the Wimp calculates the width of all text and sprite icons that it places on the icon bar.

Where the text of an icon is fixed, you should specify the icon's width to be the same as the sprite's. The Wimp then calculates the actual width to make all the text readable, both on the icon's creation and on a font change.

Where the text may change (for example if it is used to display a status), you must instead handle things yourself. You must measure the length of all potential strings using Wimp_TextOp 1 (page 5a-503), and hence find the maximum width of the icon. You must then create the icon with this width, disabling the Wimp's auto-sizing by including an 'X' in the icon's validation string. When you receive Message_FontChanged (page 5a-511) to tell you the desktop font has changed, you must recalculate the widths, and resize the icon by calling Wimp_ResizeIcon (page 5a-509).

An alternative to the above is to delete and recreate the icon each time the text changes, and rely upon the Wimp to auto-size it (ie don't include an 'X' in the icon's validation string). However, this causes much redrawing of the whole icon bar, and so is deprecated.

Tiled window backdrops

If the Wimp finds a sprite named `tile_1` in either a window's sprite area or its own (eg loaded with *IconSprites), then this is used to tile the background of a window that normally has colour 1 as its background. To improve performance the Wimp sprite pool is also searched for a sprite named `tile_1-`*`xx`*, where *`xx`* is the number of bits per pixel for the current screen mode; if one is found, this avoids the overhead of converting the tiling sprite between different pixel depths.

Tiling sprites must not have a palette.

There is a CMOS bit to enable and disable this feature; see *CMOS RAM allocation* on page 5a-73.

The Wimp's error system

Introduction

Considerable changes were made to the Wimp's error system in RISC OS 3.5. The sections below provide an overview of the changes; *Wimp_ReportError* on page 5a-495 details how the programmers' interface has changed.

General

Many applications use the Wimp's error system to relay information just as much as to raise errors, so we now refer to the dialogue boxes as *reports* rather than as error boxes. To reflect this, the title bar has been changed to say 'Message from' rather than 'Error from'.

The appearance of reports has been improved, and messages have been changed to be more helpful, consistent and user friendly. Applications can now provide more suitable wording for messages.

The 'OK' button has been changed so it instead says 'Continue', provided the calling application is aware of the new error system. (Old applications will still use an 'OK' button.). This button is the default, and can be selected by pressing the Return key. The 'Cancel' button has not changed its wording.

However, you do not have to use just these buttons, and can add to them or replace them with any number of buttons that use any words that fit. In the rare event that your buttons do not fit on the standard window, the Wimp automatically makes it wider to accommodate them; but the buttons themselves are a fixed size, at least under RISC OS 3.5 and 3.6.

New service calls make it easy to hook into your own buttons, and in particular to make your buttons themselves open reports – say to give debugging information, or extra help.

Report categorisation

RISC OS 3.5 introduces a new scheme to categorise reports. Each category uses a different sprite in the report, taken from the Wimp's `Sprites` resource file.

- A *program report* indicates an error that should not normally occur. It strongly implies that a program somewhere (whether system or application) contains a bug. The user need not know the details of the cause, although an expert user might be interested. It' s quite possible that the application will have to be terminated, or even that the machine will have to be reset.

The other types of errors are referred to as *running reports*. They are errors that, sadly, **are** to be expected in the normal running of the machine, or which have to be understood by the user. Of these:

- An *error report* indicates that something serious or unfortunate has happened, even though it might not be a program's fault. Examples include hardware faults, corrupt or absent files or discs, and running out of a resource such as memory or disc space.
- An *information report* is more an information bulletin than an error. No evasive action is typically required of the user.
- A *question report* asks a question of the user. For instance, this might be used when the user is trying to quit with unsaved data.

Backward compatibility

There are two main problems faced by applications which wish to use the new error system, and yet still work on earlier versions of RISC OS (ie 3.1 and before):

- The 'Continue' button will be labelled 'OK' under earlier versions of RISC OS, and so the text of the report needs to change
- Custom buttons are not supported under earlier versions of RISC OS.

The first problem is easily solved by using alternative files for the text of reports. You should use lines similar to the following in your !Run file:

```
Set App$Dir <Obey$Dir>
Set App$Messages <Obey$Dir>.NewMessages
RMEnsure WindowManager 3.21 Set App$Messages <Obey$Dir>.OldMessages
```

and then in your code, instead of opening <Obey$Dir>.Messages as is customary, you should open *App*$Messages.

The second problem is more involved. Let's say you wish to display a report that has 'Discard', 'Cancel' and 'Save' buttons:

- Under earlier versions of RISC OS this would need custom code.
- Using the new error system, you can display this report using the extensions to Wimp_ReportError.

  However, if you try to use the same extensions under an earlier version of RISC OS, it will ignore your custom buttons, and instead display an 'OK' button. Furthermore, when the user clicks on 'OK' a value of 1 is returned, rather than values of 3 upwards expected from your custom buttons.

The only workround is to switch conditionally between the two methods, either by use of an environment variable as above, or by examining the version number returned by Wimp_Initialise. This maintains backwards compatibility, but uses the more efficient new features when possible.

A final issue is that report categorisation is not supported by earlier versions of RISC OS, although this has no side affects on actual behaviour, just on appearance – since the old warning sprite appears for all errors.

The caret

The Wimp sets the caret to colour 11 (ie red) in 16 and 32bpp modes.

Finding other applications

Some supplied applications have been moved, for example between the RISC OS ROM image and the disc; in future releases they may move again. If your software uses another RISC OS application !*App*, it must not assume *App*'s location, but should instead find it by reading the value of the *App*$Dir system variable.

Changes to existing SWIs

Wimp_CreateWindow (page 3-87)

The new 3D window surrounds introduced in RISC OS 3.5 ignore the scroll bar inner and outer colours declared in bytes 36 and 37 of the window block. 2D windows still behave as in earlier versions of RISC OS.

Wimp_CreateIcon (page 3-93)

The (K)eys command now restricts the caret to icons in the same ESG group from RISC OS 3.5 onwards, rather than cycling through all icons – just as we advised would happen in the *RISC OS 3 Programmer's Reference Manual*.

Wimp_CreateMenu (page 3-153)

Bytes 4 - 7 of each menu item may contain a submenu pointer or window handle, or –1 if neither. The way they are distinguished changed in RISC OS 3.5:

- A submenu pointer has bit 0 clear
- A window handle has bit 0 set, but is not equal to –1.

However, you should not rely upon this in your code, as it may be subject to further change.

Wimp_ReportError (page 3-176)

This call was extended in RISC OS 3.5 to support the new types of error report. If bit 8 of the flags word passed in R1 is set, the new types are being used, and specified using both further flag bits and other parameters passed in R3 - R5.

The new flag bits in R1 are:

Bits	Meaning
8	Set ⇒ use new types of error report, as given by bits 9 - 11 and R3 - R5
9 - 11	0 ⇒ old error sprite (non classified), 1 ⇒ information report, 2 ⇒ error report, 3 ⇒ program report, 4 ⇒ question

The values passed in R3 - R5 (ignored unless bit 8 of R1 is set) are:

R3 = pointer to sprite name
R4 = pointer to sprite area, or 1 to use the Wimp sprite area
R5 = pointer to list of text for additional buttons, comma separated and control character terminated; or 0 if none

For consistent results, all sprites you use should be defined in a 16 colour mode.

If no sprite name is passed in R3, or the error is an old style one, then the Wimp tries !*app* as a sprite name. This is a desperate measure which may not internationalise well.

The strings passed in R5 are the text of additional buttons to create. If the dialogue box does not have a Continue or Cancel button (ie bits 0 and 1 of R1 are clear on entry), then the first additional button is the default one. If the first additional button is pressed, it always returns 3 in R1 – even if it is the default. Any further buttons return 4, 5, and so on. The Continue button is the rightmost one, followed by the Cancel one, followed by any additional buttons in the order they are specified, with the Describe button (if added by RISC OS – see below) appearing at the extreme left.

Serious errors

Certain potentially serious error numbers are treated slightly differently. This happens if one or more of the following are true:

- Bit 31 of the error number is set, indicating an exception such as a data abort has occurred.
- Bits 24 - 29 of the error number have the binary value 011011, meaning the error lies in a previously unused range of error numbers now reserved for program errors
- The error number is on a list hard-coded into the Wimp, specifying some 150 errors used in earlier versions of RISC OS that are now classified as program errors.

These errors are always generated as a program report. The report always has a Cancel button, but the label on it is instead Quit. The error text is replaced by '*App* may have gone wrong. Click Quit to stop *App*'. If the program did not request a Cancel/Quit button, but it is pressed, then the Wimp terminates the application without re-entering it. It does so by calling its exit handler, since the error handler may call Wimp_ReportError again, which would be confusing or may even go into an infinite loop. A Describe button is added; if this is pressed then the report is replaced by that originally provided by the application (ie the Describe button disappears).

Wimp_ReadSysInfo (page 3-216)

This call accepts the following new system information index values from RISC OS 3.5 onwards:

R0 on entry	On exit
8	R0 = font handle of desktop font, or zero if Wimp is currently using system font
	R1 = symbol font handle, or undefined if R0 = 0
9	R0 = pointer to Wimp toolsprite control block
11	R0 = maximum size of application space

DragASprite_Start (page 3-298)

This call was changed in RISC OS 3.5 so that the dragged sprite is by default dithered, and hence appears semi-transparent. A new bit has been added to the flags word in R0 to control this feature. If bit 8 is clear (as should be the case for all existing applications) then dithering occurs; if it is set then it is disabled.

Changes to existing commands

*Desktop_… (page 3-277)

The range of available *Desktop_… commands has changed in both RISC OS 3.5 and 3.6, as the range of ROM-based applications has changed. Some applications (eg !Palette) are no longer used, others have been added (eg the Display Manager), and others have moved between the ROM and the hard disc.

All such commands are – as ever – for internal use only, and so we don't list here the *Desktop_… commands available in each version of RISC OS. If you do need such a list, type `*Help Desktop_.`.

*Desktop_SetPalette (page 3-278)

This command is not available from RISC OS 3.5 onwards.

*Backdrop (page 3-293)

From RISC OS 3.5 onwards, *Backdrop supports an extra parameter: `-Remove`. Its syntax is now:

`*BackDrop [-Centre|-Scale|-Tile|-Remove] [`*`filename`*`]`

The new `-Remove` parameter removes the current backdrop.

Service Calls

Service_ErrorStarting (Service Call &400C0)

Issued immediately after Wimp_ReportError is called

On entry

R1 = &400C0 (reason code)
R2 - R7 = values of R0 - R5 (respectively) intended for Wimp_ReportError – see page 3-176 and page 5a-495

On exit

R1 preserved to pass on call
R2 - R7 = values of R0 - R5 (respectively) actually passed to Wimp_ReportError – see page 3-176 and page 5a-495

Use

This service call is issued immediately after Wimp_ReportError is called, and before Service_WimpReportError 1 (page 3-75) is issued. It allows you to change the parameters passed to Wimp_ReportError by altering the values in R2 - R7. You must not alter any memory to which these registers point on entry; you should instead make a copy of the memory, alter that, and change the relevant register to point to your copy.

If you are adding to the list of additional buttons, you must append your new buttons rather than insert them. This avoids any confusion over button numbering should other clients add their own buttons. You should obviously keep track of the position of any buttons you have added.

This service call is only issued by RISC OS 3.5 and later.

You must pass this service call on.

Service_ErrorButtonPressed (Service Call &400C1)

Issued when any button on the error report is pressed

On entry

R0 = 0
R1 = &400C1 (reason code)
R2 = button number (1 ⇒ OK, 2 ⇒ Cancel, 3 ⇒ rightmost additional button…)
R3 = pointer to button list as it appeared on the error report

On exit

R0 = 0 to return to application
R1 preserved to pass on call
R2 = button number to return (normally unchanged)

or

R0 = 1 to redisplay error report
R1 = 0 to claim call
R2 = pointer to block holding values for new report: words in same order as registers passed to Wimp_ReportError – see page 3-176 and page 5a-495

Use

This service call is issued when any button on an error report is pressed. You might use it to recognise if one of your additional buttons has been pressed by checking the button number. (Note that other clients may have added extra buttons after yours, and so the list may differ from when the initial Service_ErrorStarting call was issued. Button numbers should remain constant, though.) You can then take appropriate action, such as displaying an extra report.

This service call is only issued by RISC OS 3.5 and later.

You must claim the call if you are going to redisplay the error report.

Service_ErrorEnding (Service Call &400C2)

Issued immediately before an error report closes

On entry

R1 = &400C2 (reason code)
R2 = button number being returned to application

On exit

R1 = 0 to claim call
R2 = button number to return to application

Use

This service call is issued immediately before an error report closes, after Service_WimpReportError 0 (page 3-75) has already been issued. It allows you to alter the button number that is returned to the application that created the error report. This is only of real use if you have dealt with the error yourself in some way.

If you do change the button number, you should claim the call; otherwise you should pass it on.

This service call is only issued by RISC OS 3.5 and later.

SWI Calls

Wimp_TextOp (SWI &400F9)

Manipulates and displays text using the current desktop font

On entry

R0 = reason code and flags:

bits 0 - 7	reason code
bits 8 - 31	flags (reason code dependent)

Other registers depend upon the reason code

On exit

R0 corrupted or used to return data
Other registers typically preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI provides a number of calls to manipulate and display text using the current desktop font.

The particular action of Wimp_TextOp is given by a reason code in bits 0 - 7 of R0 as follows:

R0	Action	Page
0	Sets the colour to use for text plotting with Wimp_TextOp 2	5a-503
1	Gets the width of a string for the current desktop font	5a-504
2	Plots text on the screen using the current desktop font	5a-505

This call is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

Wimp_TextOp 0 (SWI &400F9)

Sets the colour to use for text plotting with Wimp_TextOp 2

On entry

R0 = 0 (reason code)
R1 = foreground colour (&BBGGRR00)
R2 = background colour (&BBGGRR00)

On exit

R0 corrupted
R1, R2 preserved

Use

This call sets the colour to use for text plotting with Wimp_TextOp 2 (page 5a-505). If an outline font is in use, this sets up the values for the next call to Font_Paint (page 3-437). If the system font is being used, then this sets up the colours by calling ColourTrans_SetGCOL (page 3-350), which will affect future graphics as well as text.

This call is only available from RISC OS 3.5 onwards.

Wimp_TextOp 1 (SWI &400F9)

Gets the width of a string for the current desktop font

On entry

R0 = 1 (reason code)
R1 = pointer to control character terminated string
R2 = number of characters to include, or 0 for whole string

On exit

R0 = width of string for current font, in OS units
R1, R2 preserved

Use

This call gets the width of a string for the current desktop font. The width returned is that of the first *n* characters where R2 = *n*. If there are less than *n* characters in the string or R2 ≤ 0 then the full string width is returned.

This call might be made before plotting the string in an icon, or before using Wimp_TextOp 2 (page 5a-505). For instance, it is used by the Filer when calculating the widths of the columns in a directory display.

This call is only available from RISC OS 3.5 onwards.

Wimp_TextOp 2 (SWI &400F9)

Plots text on the screen using the current desktop font

On entry

R0 = reason code and flags:
- bits 0 - 7 = 2 (reason code)
- bits 8 - 29 reserved (must be zero)
- bit 30 set ⇒ vertically justify text so baseline matches that of system font
- bit 31 set ⇒ right justify text to position given by R4 and R5

R1 = pointer to null terminated string
R2, R3 reserved (must be –1)
R4 = bottom left x coordinate, in screen OS units
R5 = bottom left y coordinate, in screen OS units

On exit

R0 corrupted
R1 - R5 preserved

Use

This call plots text on the screen using the current desktop font. If bit 31 of R0 is set, then the text is right-justified to the given position. If bit 30 is set then the text will be vertically justified so that the baseline will be the same as for the system font.

This call should be made from a redraw loop; as such Wimp_SetColour (page 3-191) or Wimp_TextOp 0 (page 5a-503) is used to determine what colours are used for the text. Because an outline font may be used, the background colour must be set, so that the antialiasing colours may be found.

This call does not preserve the current font, nor the font colours.

This call is only available from RISC OS 3.5 onwards.

Wimp_SetWatchdogState (SWI &400FA)

Sets the state of the watchdog

On entry

R0 = state (0 ⇒ disable, 1 ⇒ enable)
R1 = code word, or 0 if none

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

This call sets the state of the watchdog, and is intended for use by screenlocks and protection mechanisms.

When disabling the watchdog a code word may be supplied, in which case the watchdog may only be re-enabled by supplying the same code word. In this way, another program may not turn the watchdog back on. If R1 was zero on disabling, no code word is required when re-enabling.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

Wimp_Extend (SWI &400FB)

This call is for internal use only; you must not use it in your own code.

Wimp_ResizeIcon (SWI &400FC)

Resizes or moves an icon that has already been created

On entry

R0 = window handle (–1 for iconbar)
R1 = icon handle
R2 - R5 = new icon bounding box

On exit

R0 - R5 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call resizes an icon that has already been created. As the icon's bounding box is given, this call may also be used to move an icon.

Although general purpose, it is most likely to be used by an application needing to resize icons after a font changed message. It does not invalidate the area of the icon, although this is not necessary on a font change since the message is always followed by a redraw request.

An error is given if either of the window or icon handle are invalid.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

Messages

Wimp messages

Message_FontChanged (&400CF)

This message is used by the Wimp to inform all tasks that the desktop font has changed. This message is only used by RISC OS 3.5 and later.

Tasks that change the desktop font

Any task that changes the desktop font (see **Configure WimpFont* on page 5a-512) must call Wimp_SendMessage to request that the Wimp broadcast this message. The message itself must include no message data (ie the block is 20 bytes long).

Informing tasks of a change to the desktop font

The Wimp checks for a change in the desktop font at desktop startup, at a mode change, and whenever a task requests that this message be broadcast. It does so by examining the CMOS bits and system variables used by *Configure WimpFont (see page 5a-512).

If it detects a change it loses the current font, and attempts to find the new font with the font manager. If successful, then it selects that font; otherwise it reverts to the system font. The Wimp then broadcasts Message_FontChanged to all tasks, with one word of message data:

R1+20	Font handle, or 0 for system font

Tasks can use the font handle as they see fit.

The Wimp then issues redraw requests to all windows, so that old applications which do not understand this message will still appear correctly.

* Commands

*Configure WimpFont

Sets the configured value for the font to use on the desktop

Syntax

`*Configure WimpFont` *n*

Parameters

n	A number 0 - 15 specifying the font to use: 0 ⇒ use Wimp$Font… variables (see below) 1 ⇒ use System font 2 - 15 ⇒ use font from ResourceFS (see below)

Use

*Configure WimpFont sets the configured value for the font to use on the desktop.

A parameter of 1 sets the System font.

Parameters 2 - 15 set a font from ResourceFS, used at a size of 12 points. Starting at `Resources:$.Fonts`, every directory which has a descendant `IntMetric*` (eg. `IntMetrics`, `IntMetric0`) is numbered consecutively, starting from 2. So on a standard RISC OS 3.5 system the mapping would be:

Value	Font
2	Corpus.Bold
3	Corpus.Bold.Oblique
4	Corpus.Medium
5	Corpus.Medium.Oblique
6	Homerton.Bold
7	Homerton.Bold.Oblique
8	Homerton.Medium
9	Homerton.Medium.Oblique
10	Trinity.Bold
11	Trinity.Bold.Italic
12	Trinity.Medium
13	Trinity.Medium.Italic
14	WIMPSymbol

You must not assume this mapping, since fonts can be added to ResourceFS.

A parameter of 0 tells RISC OS to find the font and size to use from system variables:

`Wimp$Font`	the name of the font to use
`Wimp$FontSize`	the size (height) of the font in $\frac{1}{16}$ths of a point
`Wimp$FontWidth`	the width of the font in $\frac{1}{16}$ths of a point

The font size and width are both optional. If the size is unset, a value of 192 is used (ie 12 point); if the width is unset, it is the same as the size.

This command is only available from RISC OS 3.5 onwards.

Examples

`*Configure WimpFont 1`	*Use system font*
`*Configure WimpFont 8`	*Use 12pt Homerton.Medium from ResourceFS (assuming standard mapping)*
`Set Wimp$Font NewHall.Medium` `Set Wimp$FontWidth 160` `*Configure WimpFont 0`	*Set variables specifying NewHall font, and width of $\frac{160}{16}$ points (ie 10 point)* *Use system variables to set font*

*WimpKillSprite

Removes a given sprite from the Wimp's RAM sprite pool

Syntax

`*WimpKillSprite` *`sprite_name`*

Parameters

`sprite_name`	name of a sprite in the Wimp sprite pool

Use

*WimpKillSprite removes the given sprite from the Wimp's RAM sprite pool. It should be used with care, as deleting certain sprites will cause some applications to fail.

An error is given if the sprite to be removed is not in the Wimp's RAM sprite pool.

This command is only available from RISC OS 3.5 onwards.

Example

```
*WimpKillSprite file_fff
```

126 Drag An Object

Introduction and Overview

In RISC OS 3.6 the DragAnObject module was introduced. It provides SWI calls similar to those provided by the DragASprite module, save that you can use them to make the pointer drag **any** object around the screen. To do so, you must specify a SWI or a C / assembler function to render the object.

Since not all users will prefer this effect to dragging an outline – whether for aesthetics or performance – there is a bit in the CMOS RAM used to indicate their preference. (See *CMOS RAM allocation* on page 5a-73.) You should examine that bit before using this module; if it shows that the user would prefer to drag outlines, oblige them!

To drag an object:

1. Find or create a SWI or a C / assembler function to render the object to the screen. For example, you might use the SWI DrawFile_Render (page 5a-526) to render a Draw file.
2. Set up any registers / parameters you need to pass to the SWI / function; and any workspace they may point to, including – if necessary – the object itself.
3. Call the SWI DragAnObject_Start (see page 5a-516). This renders your object into its own workspace – so you can dispose of any workspace required by the rendering SWI / function whenever you like – and then starts a Wimp drag.
4. When the Wimp sends you an indication that your drag has finished, you should call the SWI DragAnObject_Stop (see page 5a-519).

SWI calls

DragAnObject_Start (SWI &49C40)

Starts dragging an object

On entry

R0 = flags
R1 = renderer called to render the object:
SWI number (R0 bit 16 = 0 on entry), or pointer to C / assembler function
R2 = pointer to block holding registers / parameters to pass to SWI / function
R3 = pointer to 16-byte block containing box
R4 = pointer to optional 16-byte block containing bounding box (see flags)

On exit

R0 - R4 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call starts dragging an object. To do so, it uses the given SWI / function to render your object twice into workspace that it claims. It then combines the two images into a single masked image, so only those pixels rendered will be used for the drag. It finally starts a Wimp drag of the masked image, and frees any workspace not needed for the drag itself.

You may dispose of any workspace used by the rendering SWI / function as soon as this call returns. If there is insufficient memory available to start the drag, the call reverts to a normal drag of a dotted outline.

The flags given in R0 have the following meanings:

Bits	Meaning
0 - 1	Horizontal location of object in box: 00 left 01 centre 10 right
2 - 3	Vertical location of object in box: 00 bottom 01 centre 10 top
4 - 5	Drag bounding box is: 00 whole screen 01 display area of window that the pointer's over 10 specified in block pointed to by R4
6	Bounding box applies to: 0 the box 1 the pointer
7	Control of drop-shadow: 0 don't do a drop-shadow 1 make a drop shadow
8	Control of dithering: 0 dither the dragged object 1 don't dither the dragged object
9 - 15	Reserved for future use – should be set to 0
16	Rendering is done by: 0 a SWI (see below) 1 a C / assembler function (see below)
17	If the renderer is a function, it is called in: 0 User mode 1 SVC mode (use for modules; also allows access to statics)
18 - 31	Reserved for future use – should be set to 0

The type of renderer is set by bit 16 of the flags:

- If the bit is clear then the renderer is a SWI. The block pointed to by R2 should be ten words long. These ten words are loaded into R0 - R9, and the SWI is then called from SVC mode.

- If the bit is set then the renderer is a C / assembler function, which is called in an APCS-conformant manner. The block pointed to by R2 should be four words long; these are loaded into R0 - R3 (known as `a1` - `a4` in the APCS) and passed as parameters to the function. R10 is set to the stack limit for a full descending stack (known as `sl` in the APCS), R13 is the stack pointer (known as `sp` in the APCS), and R14 is the link register (known as `lr` in the APCS).

  For modules, you should set bit 17 to request that the function be called in SVC mode. This also allows access to statics.

The blocks pointed to by R3 and – optionally – R4 have the following format:

Offset	Use	
0	x-low	box
4	y-low	bottom-left (x-low, y-low) is inclusive
8	x-high	top-right (x-high, y-high) is exclusive
12	y-high	

Related SWIs

DragASprite_Start (page 3-298), DragAnObject_Stop (page 5a-519)

Related vectors

None

DragAnObject_Stop (SWI &49C41)

Terminates any current drag operation, and releases workspace

On entry

—

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 terminates any current drag operation, and releases any workspace still claimed to do a drag. You should make this call when your application receives the User_Drag_Box reason code from Wimp_Poll (see page 3-112) during a drag.

Related SWIs

DragAnObject_Start (page 5a-516)

Related vectors

None

Example programs

The following code fragments show how you might call DragAnObject_Start from C:

SWI renderer

```
void start_drag(...)
{
        _kernel_swi_regs regs, render;

        /* Set up registers for Wimp_PlotIcon renderer */
        render.r[1] = (int) &icon;
        render.r[4] =0;
        render.r[5] =0;

        /* Set up registers for DragAnObject_Start... */
        regs.r[0] = some flags;
        regs.r[1] = Wimp_PlotIcon;
        regs.r[2] = (int) &render;
        regs.r[3] = (int) &bbox;

        /* ...and call it */
        _kernel_swi(DragAnObject_Start,&regs,&regs);
}
```

Code renderer

```
void _my_render(data)
{
        /* do the render */
}

void start_drag(...)
{
        _kernel_swi_regs regs;
        int render[4];

        /* Set up registers for _my_render renderer; render[0] - render[3] */
        /* will be passed to the function as parameters */
        render[0] = (int) data;         /* as required by renderer */

        /* Set up registers for DragAnObject_Start... */
        /* (tell it we're a function and a module) */
        regs.r[0] = some_flags + (1<<16) + (1<<17);
        regs.r[1] = (int) _my_render;
        regs.r[2] = (int) &render;
        regs.r[3] = (int) &bbox;

        /* ...and call it */
        _kernel_swi(DragAnObject_Start,&regs,&regs);
}
```

127 Draw file renderer

Introduction and Overview

DrawFile is a module that renders Draw files. You can do so either to the screen, or to a printer driver during printing. This makes it easy for you to support imported Draw files in your applications.

You can render a Draw file using either a SWI (DrawFile_Render; see page 5a-526) or a * command (*Render; see page 5a-531). Both provide similar facilities; in particular, they allow arbitrary transformations. The SWI DrawFile_BBox (see page 5a-528) allows you to determine a Draw file's bounding box before rendering it.

You can also declare the fonts within a Draw file when printing, without having to scan through the file yourself, by calling the SWI DrawFile_DeclareFonts (see page 5a-529).

Finally, there are service calls that the above SWIs issue if they encounter unknown objects within a Draw file. This provides a hook for modules that extend the Draw file format to support any new object types they may define. The service calls are Service_DrawObjectRender (page 5a-524) and Service_DrawObjectDeclareFonts (page 5a-525).

Technical details

Differences between DrawFile output and !Draw output

There are some small differences between the output of the DrawFile module and that of !Draw:

Text

A text line that uses a font which can't be found is rendered in the system font at a size to fit its bounding box.

Transformed text

Transformed text lines in the system font are supported.

A transformed text line that uses a font which can't be found is rendered in the system font at a size to fit its bounding box; its transformation is ignored.

Text areas

In a text area, if you change (for example) the margin size (\M command), the change doesn't take effect until the next output line. In Draw this refers to printable characters; but in DrawFile it includes colour and font change commands as well. (This is because DrawFile uses the Font Manager to remember the current font and colours.) This means that line breaks can happen at slightly different places when using DrawFile.

The following commands cause output to occur:

```
B C U V digits
```

The following do not:

```
! ; A D F L M P
```

By preceding the former with the latter, the problem can be avoided.

Sprite colours

For a sprite without a palette, the colours used are the Wimp colours, found by using Wimp_ReadPalette.

Errors

The errors the DrawFile module provides are

Error name	Error number	Meaning
Error_DrawFileNotDraw	&20C00	The file is not a Draw file (as recognised by the first 4 characters ‘D’, ‘r’, ‘a’, ‘w’).
Error_DrawFileVersion	&20C01	The file specifies a format version number which is not understood.
Error_DrawFileFontTab	&20C02	The file contains more than one font table.
Error_DrawFileBadFontNo	&20C03	A text line (or transformed text line) uses a font that is not in the font table object.
Error_DrawFileBadMode	&20C04	The file contains a sprite defined in a mode which is not recognised.
Error_DrawFileBadFile	&20C05	The size of an object in the file is larger than the size of the file allows.
Error_DrawFileBadGroup	&20C06	The size of an object in a group is greater than the size of the group allows.
Error_DrawFileBadTag	&20C07	The size of a tagged object’s data is larger than the size of the tagged object allows.
Error_DrawFileSyntax	&20C08	A text area has an illegal or unrecognised command sequence in it.
Error_DrawFileFontNo	&20C09	An attempt was made to set a font (with a \<digit> command) which had no definition (\F command).
Error_DrawFileAreaVer	&20C0A	The text area version command (\!) has specified a version which is not understood.
Error_DrawFileNoAreaVer	&20C0B	There is a text area with no version (\!) command.

Service calls

Service_DrawObjectRender (Service Call &45540)

Issued when the SWI DrawFile_Render encounters an unrecognised object

On entry

R0 = object type
R1 = &45540 (reason code)
R2 = pointer to block giving render state:

+0	pointer to unknown object (see *Objects* on page 4-465 and 5a-665)
+4	pointer to Draw file data, as passed to DrawFile_Render
+8	pointer to font table object, or 0 if none found yet
+12	flags, as passed to DrawFile_Render
+16	pointer to transformation matrix, as passed to DrawFile_Render
+20	pointer to clipping rectangle, as passed to DrawFile_Render
+24	flatness, as passed to DrawFile_Render
+28	pointer to error block, or 0 if no error yet

On exit

R1 = 0 if claimed, otherwise preserved

Use

This service call is issued when the SWI DrawFile_Render encounters an object with a type it doesn't recognise, and so cannot process. If a module recognises the unknown object type, it should claim the service call and itself render the object.

If the module encounters an error during rendering, it should examine the error pointer word in the passed render state block:

- If the word is zero, the declaring module should store its own error pointer in the word.
- If the word is non-zero, it is already storing an earlier error pointer, which you should not overwrite.

The DrawFile module attempts to render all objects. When it has finished it examines the error pointer word, and if it is non-zero returns the stored error.

Service_DrawObjectDeclareFonts (Service Call &45541)

Issued when the SWI DrawFile_DeclareFonts encounters an unrecognised object

On entry

R0 = object type
R1 = &45541 (reason code)
R2 = pointer to declare font state block

+0	pointer to unknown object (see *Objects* on page 4-465 and 5a-665)
+4	pointer to Draw file data, as passed to DrawFile_DeclareFonts
+8	pointer to font table object, or 0 if none found yet
+12	flags, as passed to DrawFile_DeclareFonts
+16	pointer to error block, or 0 if no error yet

On exit

R1 = 0 if claimed, otherwise preserved

Use

This service call is issued when the SWI DrawFile_DeclareFonts encounters an object with a type it doesn't recognise, and so cannot process. If a module recognises the unknown object type, it should claim the service call and itself declare any fonts in the object.

If the module encounters an error while declaring the fonts, it should examine the error pointer word in the passed font state block:

- If the word is zero, the declaring module should store its own error pointer in the word.
- If the word is non-zero, it is already storing an earlier error pointer, which you should not overwrite.

The DrawFile module attempts to declare fonts for all objects. When it has finished it examines the error pointer word, and if it is non-zero returns the stored error.

SWI calls

DrawFile_Render (SWI &45540)

Renders a Draw file to the screen

On entry

R0 = flags:
bit 0 set ⇒ render the bounding boxes around objects as dotted red rectangles
bit 1 set ⇒ do not render the objects themselves
bit 2 set ⇒ R5 is used as the flatness parameter
R1 = pointer to Draw file data
R2 = size of Draw file data, in bytes
R3 = pointer to transformation matrix, or 0 for identity
R4 = pointer to clipping rectangle in OS units, or 0 if no clipping rectangle set up
R5 = flatness with which to render lines (if bit 2 of R0 set on entry)

On exit

R0 - R5 preserved

Interrupts

Interrupt status in undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call renders a Draw file to the screen. Its position is given by the x- and y-translations in the transformation matrix, which uses the same format as Draw (see *Transformation matrix* on page 3-536).

Hence to render a non-rotated 1:1 Draw file at screen coordinates (x, y) OS units, the transformation matrix is:

$$\begin{pmatrix} 1 << 16 & 0 \\ 0 & 1 << 16 \\ 256 \times x & 256 \times y \end{pmatrix}$$

The effects of calling the module with the matrix not of the form:

$$\begin{pmatrix} +f & 0 \\ 0 & +f \\ x & y \end{pmatrix}$$

(ie a translation and a magnification) should not be relied on for underlined text.

If no transformation matrix is given (ie R3 = 0), the unit matrix is used, and so the Draw file is rendered with its bottom left corner at screen coordinates (0, 0).

The clipping rectangle is typically a redraw rectangle returned by the Wimp on a redraw window request. If R4 = 0, then the whole Draw file is rendered. If non-zero, only objects which intersect the clipping rectangle are rendered.

All output calls used when rendering are ones that the printer drivers handle correctly, so you can also use this call to output Draw files when printing.

Just as for all other screen output calls, if you make this call in a Wimp redraw loop (ie after calling Wimp_RedrawWindow) you cannot use Wimp_ReportError to report any error that is returned – since this might lead to an infinite loop of error boxes and redraws of the rectangle covered by the error box. This restriction does not apply to printing redraw loops (ie after calling PDriver_DrawPage).

Related SWIs

DrawFile_BBox (page 5a-528), DrawFile_DeclareFonts (page 5a-529)

Related vectors

None

DrawFile_BBox (SWI &45541)

Returns the bounding box (in Draw units) a given Draw file will occupy

On entry

R0 = flags: all bits reserved (must be 0)
R1 = pointer to Draw file data
R2 = size of Draw file data, in bytes
R3 = pointer to transformation matrix, or 0 for identity
R4 = pointer to 4 word buffer to hold the bounding box of the Draw file (x0, y0, x1, y1) in internal Draw units

On exit

R0 - R4 preserved

Interrupts

Interrupt status in undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the bounding box (in Draw units) the given Draw file would occupy, were it to be plotted with the given transformation.

Related SWIs

DrawFile_Render (page 5a-526), DrawFile_DeclareFonts (page 5a-529)

Related vectors

None

DrawFile_DeclareFonts (SWI &45542)

Declares all fonts in a Draw file by calling PDriver_DeclareFont

On entry

R0 = flags:
bit 0 set ⇒ do not download font (passed to PDriver_DeclareFont)
R1 = pointer to Draw file data
R2 = size of Draw file data, in bytes

On exit

R0 - R2 preserved

Interrupts

Interrupt status in undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call declares all fonts in a Draw file by calling PDriver_DeclareFont (see page 3-648). If the printer driver you are using supports PDriver_DeclareFont you should call this SWI once for each Draw file to be printed, at the point where you would normally declare fonts (see *Declare the fonts your document uses* on page 3-569). This saves you having to scan the Draw file yourself to see which fonts it uses. It is your responsibility to make the final call to PDriver_DeclareFont to indicate the end of the list of fonts.

All fonts are declared as ‘kerned’, since this includes the non-kerned case.

Related SWIs

DrawFile_Render (page 5a-526), DrawFile_BBox (page 5a-528)

Related vectors

None

* Commands

*Render

Displays the contents of a Draw file

Syntax

```
*Render [-file] filename [m00 m01 m10 m11 m20 m21]
        [-bbox] [-suppress] [-flatness flatness]
```

Parameters

`[-file] filename`	a valid pathname specifying the Draw file to be rendered
m00, *m01*, *m10*, *m11*	four decimal numbers giving the first four elements of the transformation matrix, which specify the transformation
m20, *m21*	two decimal numbers giving the other two elements of the transformation matrix, which specify the translation in OS units
`-bbox`	render the bounding boxes around objects as dotted red rectangles
`-suppress`	suppress the rendering of the objects themselves
`-flatness flatness`	a decimal number giving the flatness used to render curved paths, in OS units

Use

*Render displays the contents of a Draw file. You can optionally transform the output with a transformation matrix, render the bounding boxes around objects as dotted red rectangles, suppress the rendering of the objects themselves, and change the flatness used to render curved paths. In doing so, the file is checked for consistency.

Note that you must quote any negative arguments; see the example below.

Example

```
*Render adfs::MHardy.$.DrawFile 0 "-1.5" 1.5 0 0 800 -bbox -flatness .5
```

Related commands

None

*Render

128 RISC OS boot applications

Introduction and Overview

A boot application sets the machine up whenever the computer is reset, giving users and applications control over the start up, configuration and use of the desktop. It also works interactively when a user double clicks on its icon.

RISC OS 3.5's boot application is named !Boot. RISC OS 3.6 introduced a wider range of boot applications (eg !Boot for booting from disc, !ArmBoot and !ShareBoot for booting from a remote machine or server on a network). You can use boot applications to boot older machines.

A boot application provides:

- startup files that applications can modify
- facilities to link applications into the Apps icon's directory display
- desktop boot saving
- locations for saving application-specific choices
- computer configuration using !Configure, which is held as a sub application
- hard disc locking with a password, as a part of !Configure
- !System, !Scrap and !Fonts as sub applications that are unseen by the user.

This chapter details the facilities provided by boot applications for integrating applications into the desktop, and that will continue to be supported in the future. Unless otherwise stated, all such facilities are available from RISC OS 3.5 onwards.

Although a search through a boot application will doubtless reveal features that are not documented here, you must not use them if you wish your application to work under future versions of RISC OS.

The user interface is described in the *RISC OS 3 User Guide*.

Technical details

Writing to a boot application

Amongst other things, this chapter tells you ways you can add to the files in a boot application. When doing so you must be aware that you may not have write permission, especially if the boot application is on a remote file server. If you try to write to a boot application, and the write fails, your software should fail gracefully, giving a suitable error message.

Starting the boot application

The kernel determines which boot application to run depending on the configured Boot state, filing system, drive/file server, and so on. It then tries to find the boot application. If it fails, it saves the resultant error message in the system variable Boot$Error; if it succeeds, it runs the boot application.

Environment set up

The Boot$… variables

Once a boot application has been found and run, it sets up a number of system variables as it first starts:

Variable	Comments
Boot$Dir	The boot application's directory
Boot$OSVersion	The major version of RISC OS on the booting computer: for example '200', '310', '350', or '360'
Boot$Path	Comma separated list of boot application directory(s), each with a trailing '.'
Boot$State	The stage of booting: 'commands' or 'desktop'
Boot$ToBeLoaded	PreDesk directory (see page 5a-540)
Boot$ToBeTasks	Tasks directory (see page 5a-543)
Boot$Unique	'Local' if the boot file is local; a unique identifier for the machine if the boot file is remote

(The RISC OS 3.5 !Boot application does not set the Boot$State and Boot$Unique variables.)

The Choices$… variables

It then uses these variables to set up other variables giving the location(s) of Choices directory(s), which are used by applications – including the Boot application itself – to store start-up files, user preferences, and so on:

Variable	Comments
Choices$Dir	Most recently used Choices directory
Choices$Path	Comma separated list of Choices directory(s) from which to read, each with a trailing '.'
Choices$Write	Choices directory to which to write

Choices directories

A boot application may have multiple Choices directories, and so store different choices for different machines, such as remote network stations. Each Choices directory can have the same structure beneath it, varied as required by the different machine(s) using each one.

RISC OS 3.6's network boot applications set the Choices$… variables to use the directory `Boot:MchConfig.<Boot$Unique>` out of preference, if it exists; failing that, to use `Boot:Utils.RO<Boot$OSVersion>Hook` if that exists, or otherwise to use `Boot:Choices`. Remote network stations will thus look for machine-specific choices; then either for RISC OS version-specific choices, or for system-wide choices. Network managers can hence use these different directories to support a mix of versions of RISC OS, and to provide machine-specific exceptions to the general setup.

For example, with the boot application `Net::Server.$.!ArmBoot`, these directories might be named:

- *Machine-specific choices:*
  `Net::Server.$.!ArmBoot.MchConfig.Stn128!005` *or:*
  `Net::Server.$.!ArmBoot.MchConfig.E1268AFB` *etc*
- *RISC OS version-specific choices:*
  `Net::Server.$.!ArmBoot.Utils.RO200Hook` *or:*
  `Net::Server.$.!ArmBoot.Utils.RO310Hook` *etc*
- *System-wide choices:*
  `Net::Server.$.!ArmBoot.Choices`

You **must not** access the Choices directories by evaluating their full pathnames yourself, as the internal structure of boot applications is liable to change in future versions. You **must** instead use the methods described in the sections below.

Loading CMOS

After setting these system variables, the next thing of interest the boot application does is to look for the file:

```
<Boot$Dir>.MchConfig.<Boot$Unique>.!RO<Boot$OSVersion>CMOS
```

If it exists, the boot application uses the *LoadCMOS command to load the contents of the file into CMOS RAM.

Files used before the desktop is started

The boot application then runs things before the desktop is entered. It uses two locations for this:

- The PreDesktop file is an Obey file. It creates various useful aliases for common tasks.

  You can add to this file as necessary, in the manner described below. If you only need to use an Obey file, this is the preferred method to use.
- The PreDesk directory holds files and directories all of which are run after the PreDesktop file.

  Adding files and directories here gives you much more flexibility over how they are run. You should use this method if your needs are not met by the PreDesktop file.

For full details, see the sections below.

The PreDesktop file

The PreDesktop file contains the command line setup sequence. It gets invoked using Obey -c where possible, so that filing system or network software can be reloaded during its execution.

Accessing the file

The PreDesktop file must always be accessed as:

```
<Choices$Write>.Boot.PreDesktop
```

This is to preserve future compatibility, should the structure of boot applications change.

Format of the file

The file has been divided into well defined sections for ease of maintenance, and to make it easy for scripts to install and remove application-specific entries. Each section starts with a header:

```
|Start Company Application Version Section
```

and ends with a footer:

```
|End
```

As an example, the Aliases section supplied by !Boot in RISC OS 3.5 looks like this:

```
|Start Acorn !Boot 0.25 Aliases
commands and comments…
|End
```

Scripts that scan the file should be extremely lax in what they accept. They should accept any amount of white space between each element (including before and after the '|' that introduces the header/footer), ignore case, and ignore the version numbers used by other applications.

When writing to the file scripts should use the exact syntax in the above example; see *Adding to the file* on page 5a-537.

Sections

The sections supplied by a boot application are:

- Comments
- Aliases
- Paths
- Options
- ResApps
- Miscellaneous options.

These are described in more detail below, starting with *Comments* on page 5a-538.

Adding to the file

Wherever possible, you should split anything you add into the same sections as above, each of which includes a header and footer. Each application should provide a script to install its sections. For each section, the script must:

1 Search for the boot application's corresponding section.

2 Add the new section after the boot application's corresponding section.

So a finished file might look like this:

```
|Start Acorn !Boot 0.25 Comments
comments…
|End
|Start Acorn !Boot 0.25 Aliases
commands and comments…
|End
```

```
|Start Acorn !Boot 0.25 Paths
commands and comments…
|End
|Start Acorn !Boot 0.25 Options
commands and comments…
|End
|Start MySWHouse !MyApp 1.10 Options
commands and comments…
|End
|Start Acorn !Boot 0.25 ResApps
commands and comments…
|End
|Start Acorn !Boot 0.25 Miscellaneous options
commands and comments…
|End
```

It is courteous to also supply a script to remove the entries.

Comments

The function of this section should be obvious!

Aliases

This section sets aliases.

A boot application's section adds several aliases that you may find useful yourself:

Alias

The first alias set is for Alias itself, so that the following command:

```
Alias alias command
```

sets the alias *alias* for the command *command*.

Unalias

This makes the command Unalias to remove an alias:

```
Unalias alias
```

Path

The next alias gives a convenient way to set setting of paths:

```
Path path full_path
```

so you can refer to a full pathname *full_path*rest using the shorthand *path*:rest. For example:

```
Path lib ADFS::HardDisc4.$.Library.
```

would enable the following convenient commands:

```
*Dir lib:
/lib:cc
```

PathMacro

PathMacro works similarly to Path, except the system variable set is a macro variable. For example:

```
Set Thing$Dir <Obey$Dir>
PathMacro Thing <Thing$Dir>.
```

To enable Thing: to be a reference to <Thing$Dir>.

Paths

This section is used to set standard paths and directories.

Run$Path is defined here to include the Library directory held within the boot application. This allows you to use the various commands held in the library, and defined in * *Commands* on page 5a-545.

Options

This section has been set aside for options that do not have any other place to be set.

ResApps

This section uses the AddApp library command (see page 5a-545) to register applications with ResourceFS for display in Resources:$.Apps.

A boot application's ResApps section registers all applications in Boot:^.Apps:

```
AddApp Boot:^.Apps.!*
```

Miscellaneous options

This section is used for any setup that does not obviously belong in any of the above sections. An example might be loading and binding a novel system beep.

The PreDesk directory

Accessing the directory

The PreDesk directory must always be accessed as:

```
<Boot$ToBeLoaded>
```

This is to preserve future compatibility, should the structure of boot applications change.

Adding files and subdirectories

Your application !*App* may add a single file or subdirectory named *App*. You should only do so if the PreDesktop file does not meet your needs, since if too many applications use this directory, it may become full.

Your application may modify its own file(s) as it sees fit.

Action taken on files and subdirectories

RISC OS 3.60

Under RISC OS 3.60, the files held within the PreDesk directory are acted on as follows:

- Any files of type Obey are run using *Obey -c, or *Obey for versions of RISC OS where the -c flag is not supported.
- Any files of type Absolute are run.
- Any files of type Sprite are loaded using *IconSprites.
- Any files of type Module are loaded using *RMLoad.
- Any files of type BASIC are run using *BASIC -quit.
- Any files of type Utility are run.

All other files are loaded using *Load.

Then any directories are run; this searches for the file !Run in the directory, and runs it if found.

RISC OS 3.50

Under RISC OS 3.50, the files and directories held within the PreDesk directory are acted on in the following order:

1 Any files of type Module are run using *RMLoad.

2 Any files of type Sprite are run using *IconSprites.

3 Any files of type Obey are run using *Obey -c.

4 Any directories are run using *Run; this searches for the file `!Run` in the directory, and runs it if found.

All other files are ignored.

Files used once the desktop is started

Desktop saving

The method for saving the desktop from the Task Manager remains the same as before, and is still the preferred way for applications to set themselves up and start. You should continue to use desktop saving provided it meets your needs.

If it does not meet your needs, you should read the sections below.

Other files

Earlier sections described how boot applications provide a file and a directory that are used to run things before the desktop is entered. They provide a similar file and directory that your application can use to start itself and/or any associated tasks:

- The Desktop file is a file of type Desktop, run as the desktop is entered. It boots important system resources.

  You can add to this file as necessary, in the manner described below. If you only need to use a Desktop file, but cannot use the Task Manager's desktop boot file, then it is preferable to use this file rather than adding to the Tasks directory.

- The Tasks directory holds files all of which the Desktop file starts as Wimp tasks, including the desktop boot file saved from the Task Manager.

  You should only add files here if your needs are met neither by the Task Manager's desktop boot file, nor by the Desktop file.

For full details, see the sections below.

The Desktop file

Accessing the file

The Desktop file must always be accessed as:

```
<Choices$Write>.Boot.Desktop
```

This is to preserve future compatibility, should the structure of boot applications change.

Format of the file

The file has the same format as the PreDesktop file. It is split into sections using headers and footers with the same syntax. See *Format of the file* on page 5a-536.

Sections

The sections supplied by a boot application are:

- Auto tasks
- Completion

The *Auto tasks* section is described in more detail below.

Adding to the file

You should add sections to the Desktop file in just the same way as for the PreDesktop file. See *Adding to the file* on page 5a-537.

Note that you should only add an *Auto tasks* section. There should be no *Completion* section in the file apart from that provided by the boot application itself.

Auto tasks

This section boots all the system resources held in `Boot:Resources`:

```
Repeat Filer_Boot Boot:Resources -Applications -Tasks
```

This includes such things as !System, !Scrap and !Fonts.

It then runs all the files in the Tasks directory as Wimp tasks:

```
Repeat Filer_Run <Boot$ToBeTasks> -Tasks
```

The Tasks directory

Accessing the directory

The Tasks directory must always be accessed as:

```
<Boot$ToBeTasks>
```

This is to preserve future compatibility, should the structure of boot applications change.

Adding files and subdirectories

Your application !*App* may add a single file or subdirectory named *App*. You should only do so if the Desktop file does not meet your needs, since if too many applications use this directory, it may become full.

Your application may modify its own file(s) as it sees fit.

Action taken on files

The files and directories held within the Tasks directory are run using Filer_Run.

Storing application choices

Your application !*App* can create its own Choices directory, and use it to store user preferences.

Accessing the directory

Your application !*App* must always access its Choices directory as:

```
<Choices$Write>.App
```

This is to preserve future compatibility, should the structure of boot applications change.

Adding files and subdirectories

Your application !*App* may add any files or subdirectories it needs to. It may modify its own file(s) as it sees fit.

Changes to existing * Commands

*Logon (page 2-386)

You should note that from RISC OS 3.6 onwards, the Boot application aliases the *Logon command to *SafeLogon, described on page 5a-554. (Although this is not strictly a change to the *Logon command, most people will see it as such.)

In the unlikely event you need to force the use of the standard *Logon command, you must do so by using the '%' character to skip alias checking (see *CLI effects* on page 1-956), rather than by unsetting the variable Alias$Logon. This ensures that the change only applies to your command line, and does not alter the environment other programs expect to find.

The BootCommands module

A boot application uses various commands not provided by RISC OS 3.1 or earlier.

In RISC OS 3.5, these are provided by the boot application's Library subdirectory, which is added to the run path when the boot application is first run. In RISC OS 3.6 most of these are instead provided by a new module named BootCommands.

The advantage of the BootCommands module is that it avoids the need to load the commands over the network if a station is using a remote boot application. Booting is thus made faster.

* Commands

*AddApp

Adds entries in Resources:$.Apps for all applications matching a wildcard pattern

Syntax

`*AddApp [`*`directory`*`.]`*`pattern`*

Parameters

`directory`	a valid pathname specifying a directory
`pattern`	wildcarded pattern to match

Use

*AddApp adds entries in Resources:$.Apps for all applications matching the wildcard pattern in the given directory, or in the current directory if none is specified. If nothing matches the pattern, no error is generated; the command just returns.

You must not use this command to add applications that are already held in ResourceFS.

This command is provided either by the boot application's Library subdirectory (which is added to the run path when the boot application is first run), or by the BootCommands module added in RISC OS 3.6.

Example

```
*AddApp adfs::MHardy.$.MyApps.*
```

Related commands

None

*AppSize

Moves memory into or out of the RMA

Syntax

```
AppSize size[K]
```

Parameters

size[K]	number of (kilo)bytes of memory desired for applications

Use

*AppSize moves memory into or out of the RMA, attempting to move the difference between the current size of application workspace and the given desired size. In RISC OS 3.1 and earlier, the memory was transferred to/from the application workspace (hence the name of the command); from RISC OS 3.5 onwards, memory is transferred to/from the free pool.

This command is used at startup to shrink the RMA to its smallest possible size by setting the desired application size to a large value, and should not be used by other applications.

This command is provided either by the boot application's Library subdirectory (which is added to the run path when the boot application is first run), or by the BootCommands module added in RISC OS 3.6.

Example

```
AppSize 514000K
```

Related commands

None

*Do

Passes a command to XOS_GSTrans, and then passes it to the CLI

Syntax

`*Do` *`command`*

Parameters

`command`	command to have GSTrans'd before execution

Use

*Do passes a command to XOS_GSTrans, and then passes it to the CLI.

It is useful when the command being invoked does not itself GSTrans its parameters, but you wish to pass parameters using GS string format (eg system variables). For more details, see *GS string operations* on page 1-454.

This command is provided either by the boot application's Library subdirectory (which is added to the run path when the boot application is first run), or by the BootCommands module added in RISC OS 3.6.

Example

`*Do BadCmd <Obey$Dir>` *Expands Obey$Dir before calling BadCmd*

Related commands

None

*FontMerge

Merges new fonts into an existing !Fonts directory

Syntax

`FontMerge` *`source`* [*`destination`*]

Parameters

`source`	source directory of fonts to merge
`destination`	destination directory of fonts to merge

Use

*FontMerge merges new fonts into an existing !Fonts directory. The first thing it does is to work out the destination for the merge.

If no destination is given, the third-from-last element of Font$Path is used. This may seem a bit strange, but consider what Font$Path will look like:

```
Font$Path(Macro):
ADFS::HardDisc4.$.!Boot.Resources.!Fonts.,<Font$Prefix>.,Resources:$.Fonts.
```

The last element is in the Resource filing system, which cannot be used as the destination. The next-to-last element is <Font$Prefix>; this is provided for backwards compatibility, so it is not a good idea to use it as the destination. The third-from-last element is therefore the one used.

*FontMerge can automatically create and use an overflow directory should the original destination become full. For a directory !Fonts, the overflow directories are !Fonts1, !Fonts2, and so on. *FontMerge checks for the presence of such overflow directories, and uses the highest numbered one as the initial destination.

Once *FontMerge has worked out the destination, it merges the fonts, creating overflow directories as necessary. It automatically processes font messages files, generating them for all languages given in the source and destination.

*FontMerge can be run from desktop applications. It initialises itself as a Wimp task to generate Wimp error boxes if it has an error; it calls Hourglass_Percentage as it does the merge.

This command is provided by the boot application’s Library subdirectory (which is added to the run path when the boot application is first run). *FontMerge is a directory, and should be left as such. This is to enable *FontMerge to be localised for a particular

country simply by replacing the messages file inside the FontMerge directory. Even though *FontMerge is a directory and not a file, you use it just like any other command line program.

Example

```
*FontMerge adfs::FontVendor.$.!Fonts
```

Related commands

None

*IfThere

Checks for the presence of a given object, and executes one command if it exists, or another if it does not

Syntax

```
*IfThere object_spec Then true_command [Else false_command]
```

Parameters

object_spec	a valid (wildcarded) pathname specifying a file or directory
true_command	command to execute if *object_spec* is matched
false_command	command to execute if *object_spec* is not matched

Use

*IfThere checks for the presence of the given object, and executes the true command if it exists, or the optional false command if it does not.

The check is done using OS_File 17 (page 2-38). Note that non-files (eg directories and partitions) will still cause the true command to execute.

This command is provided either by the boot application's Library subdirectory (which is added to the run path when the boot application is first run), or by the BootCommands module added in RISC OS 3.6.

Example

```
*IfThere adfs::MHardy.$.Run Then Delete adfs::MHardy.$.Run
```

Related commands

None

*LoadCMOS

Loads a file into the computer's CMOS RAM

Syntax

`*LoadCMOS` *`filename`*

Parameters

`filename`	a valid pathname specifying a file

Use

*LoadCMOS loads a file into the computer's CMOS RAM, preserving only the station number, the current year, and the DST flag. All other configured values are replaced by those stored in the file.

This command is used by boot applications to load a station's CMOS RAM at startup time, thus ensuring the machine is always in the same state. The boot application searches for the file:

`<Boot$Dir>.MchConfig.<Boot$Unique>.!RO<Boot$OSVersion>CMOS`

and, if it finds the file, uses this command to load it. Note that the location of saved CMOS files is subject to change in future versions of boot applications.

This command is provided by the BootCommands module added in RISC OS 3.6. Unlike most other commands documented in this chapter, it is not a standard part of the RISC OS 3.5 boot application.

Example

```
*LoadCMOS adfs::MHardy.$.Safe.MyCMOS
```

Related commands

None

*Repeat

Scans a given directory, applying a command to everything it finds

Syntax

```
*Repeat command directory [-Directories] [-Applications]
  [-Files|-Type file_type] [-CommandTail cmdtail] [-Tasks]
```

Parameters

command	command to apply to objects in the given directory
directory	a valid pathname specifying a directory
-Directories	apply the command only to directories
-Applications	apply the command only to applications
-Files	apply the command only to files
-Type	apply the command only to files of type *file_type*
file_type	a number (in hexadecimal by default) or text description of the file type to match. The command *Show File$Type* displays a list of valid file types.
-CommandTail	postfix the found object with *cmdtail*
cmdtail	command tail to apply to objects in the given directory
-Tasks	apply the command as a Wimp task

Use

*Repeat scans the given directory applying a command to everything it finds, within the limits of the other parameters. The command executed is:

command found_object [*cmdtail*]

This utility does not recurse. Only those objects identified at the top level have the command applied to them.

This command is provided either by the boot application's Library subdirectory (which is added to the run path when the boot application is first run), or by the BootCommands module added in RISC OS 3.6.

The Library-based version uses the Scrap directory, and hence there must be some free space on the file system holding !Scrap for it to work. The BootCommands version does not have this limitation.

Example

```
*Repeat Filer_Boot Boot:Resources -Applications -Tasks
```

Related commands

None

*SafeLogon

Logs you on to a file server if you are not already logged on

Syntax

```
*SafeLogon [[:]file_server_number|:file_server_name] user_name [[:Return]password]
```

Parameters

file_server_number	the file server number to log on to
file_server_name	the file server name to log on to
user_name	as issued by the network manager
password	as set by the user

Use

*SafeLogon logs you on to a file server if you are not already logged on.

The command first checks to see if the current temporary filing system is NetFS, and the given user is already logged on to the given file server; if so, the command exits immediately. Otherwise the command passes on a *Logon command to the current temporary filing system, leaving the command line tail unaltered.

This means that – unlike *Logon – *SafeLogon will not log you off a file server, and then immediately log you back on.

This command is provided by the BootCommands module added in RISC OS 3.6. Unlike most other commands documented in this chapter, it is not a standard part of the RISC OS 3.5 boot application.

Example

```
*SafeLogon :fs guest
```

Related commands

*Logon

129 The colour picker

Introduction and Overview

The new hardware supported by RISC OS 3.5 supports a much greater pixel depth than previous versions, and can display up to 16 million colours. The colour picker module is a utility that allows users to pick a colour from this immense choice. This utility should be used by all applications that need to choose colours.

This chapter describes how a client application and the colour picker module interact.

Unless stated otherwise, all the facilities described in this chapter are available from RISC OS 3.5 onwards.

Terminology used

The *colour picker* module provides a *colour picker dialogue* for applications to use; different types of dialogue are available. The dialogue is not a Wimp task. The colour picker makes use of the Wimp filter mechanism to receive events for its dialogues.

A colour picker *client* is an application which use the colour picker. All clients must be Wimp tasks.

A *colour descriptor* is a structure giving the full details of a colour. It is defined on page 5a-558.

Technical details

How the colour picker works

The colour picker works as follows:

- The client application communicates with the colour picker by calling SWIs.
- Whenever a client opens a new colour picker dialogue, the colour picker module installs a Wimp pre-filter and post-filter box around that client application.
- The colour picker module then maintains the colour dialogue by intercepting Wimp events directed to it, and passing the user's colour choices to the client using Wimp messages.
- Once the dialogue is opened, it is identified in all SWIs and messages by a handle. This avoids confusion if multiple clients are using the colour picker at once.

Typical usage

From the client's point of view, a typical colour selection looks like this:

```
Client receives colour selection request by user
Client prepares structure describing dialogue
Client issues the SWI ColourPicker_OpenDialogue and resumes polling the Wimp
User makes colour selection
Client receives the message Message_ColourPickerColourChoice
Client applies colour information
Client receives the message Message_ColourPickerCloseDialogueRequest
Client issues the SWI ColourPicker_CloseDialogue
```

SWIs and messages used

SWIs

The full range of SWIs that clients may use are as follows:

- ColourPicker_OpenDialogue (page 5a-563) – Creates and opens a colour picker dialogue for a client.
- ColourPicker_CloseDialogue (page 5a-566) – Closes a colour picker dialogue which is in progress.
- ColourPicker_UpdateDialogue (page 5a-567) – Updates some or all of the contents of a colour picker dialogue.
- ColourPicker_ReadDialogue (page 5a-569) – Reads the current state of a colour picker dialogue without changing it.

- ColourPicker_HelpReply (page 5a-572) – Makes the colour picker respond to a Message_HelpRequest with its own help text.

Messages

The messages that the client may receive are:

- Message_ColourPickerColourChoice (page 5a-575) – issued whenever the user makes a definite choice of colour.
- Message_ColourPickerColourChanged (page 5a-575) – optionally issued when the colour displayed in the dialogue changes.
- Message_ColourPickerCloseDialogueRequest (page 5a-576) – optionally issued when the user dismisses the dialogue.
- Message_ColourPickerOpenParentRequest (page 5a-576) – issued when the user opens a toolbox dialogue's parent by clicking Adjust or Shift-Adjust on its Close icon.
- Message_ColourPickerResetColourRequest (page 5a-576) – issued when the user requests the colours be reset to those currently in effect.

  This message is not issued by RISC OS 3.5.

Dialogue types

When the client calls ColourPicker_OpenDialogue (page 5a-563) to create and open the colour picker dialogue, it can choose between different types of dialogue:

- A *normal dialogue* has **OK** and **Cancel** buttons, and issues Message_ColourPickerColourChoice (page 5a-575) when **OK** is used.
- A *toolbox dialogue* has no **OK** and **Cancel** buttons, but has Back and Close icons on its window. It never issues Message_ColourPickerColourChoice; the client can call ColourPicker_ReadDialogue (page 5a-569) to read the colour when it needs to, or monitor the colour continuously by making the dialogue issue Message_ColourPickerColourChanged (page 5a-575).
- A *menu dialogue* is like a normal dialogue, but the ColourPicker opens it using Wimp_CreateMenu; it is therefore automatically closed by the Wimp when the user clicks elsewhere.
- A *submenu dialogue* is like a menu dialogue, except that it is attached to an open menu tree, and hence created using Wimp_CreateSubMenu.

Transient dialogues

Because the closing of menu and submenu dialogues is handled by the Wimp rather than the colour picker, they are classed as *transient dialogues*.

Colour descriptors

Colours are passed to and from the colour picker as a colour descriptor, which is a structure of two or more words.

The colour is always held in a word as a 24 bit RGB value; the simplest form of a colour descriptor has no extra information. However, the descriptor may also hold data giving a colour model, the colour value as represented in that model, and optional extra data. Clients may store the entire colour descriptor, and make full use of the information it stores.

A colour descriptor has this structure:

Offset	Data
0	0
1	red value (0 - &FF)
2	green value (0 - &FF)
3	blue value (0 - &FF)
4	a word giving the size of the optional extension to the block, in bytes

The optional extension consists of:

Offset	Data
8	colour model number: 0 ⇒ RGB, 1 ⇒ CMYK, 2 ⇒ HSV
12	colour model dependent data (see below)

An application may treat the colour descriptor as a self contained block to be stored away, and retrieved for use later with the colour picker.

Colour model dependent data

All colour model dependent data uses fixed point 32 bit numbers, with 16 bits below the point, and 16 bits above the point. Offsets given below are relative to the start of the colour model dependent data.

RGB (model number 0)

The extra data is:

Offset	Data
0	Red value
4	Green value
8	Blue value

All values should be in the range &00000000 - &00010000.

CMYK (model number 1)

The extra data is:

Offset	Data
0	Cyan value
4	Magenta value
8	Yellow value
12	Key (black) value

All values should be in the range &00000000 - &00010000.

HSV (model number 2)

The extra data is:

Offset	Data
0	Hue angle
4	Saturation percentage
8	Value percentage

The Hue should be in the range &00000000 - &0167FFFF. The Saturation and Value should be in the range &00000000 - &00010000.

Wimp events and the client

Wimp events directed to the colour picker are also sent to the client. They can be distinguished by the window handle in the event block.

These events have already been fully processed by the colour picker. They are sent to the client merely as a ‘hook’ for unusual circumstances where special action is required. This won’t normally be necessary, and the client should ignore these events, just as it should any events it does not understand.

For example, if it is told of a request for help by a User_Message event (of type Message_HelpRequest) that has the colour picker’s window handle, it should not supply help, since the colour picker will already have done so.

Service Calls

Service_ColourPickerLoaded (Service Call &93)

This service call is for internal use only; you must not use it in your code.

SWI calls

ColourPicker_RegisterModel (SWI &47700)

This call is for internal use only; you must not use it in your own code.

ColourPicker_DeregisterModel (SWI &47701)

This call is for internal use only; you must not use it in your own code.

ColourPicker_OpenDialogue (SWI &47702)

Creates and opens a colour picker dialogue for a client

On entry

R0 = flags:
bits 0, 1 = dialogue type:
0 ⇒ *normal* dialogue, 1 ⇒ *menu* dialogue
2 ⇒ *toolbox* dialogue, 3 ⇒ *submenu* dialogue
all other bits reserved (must be set to 0)
R1 = pointer to a colour picker block (see below)

On exit

R0 = dialogue handle
R1 = window handle

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 creates and opens a colour picker dialogue for a client, so that a user can choose a colour. The flags in R0 set the type of dialogue (see *Dialogue types* on page 5a-557), and hence whether or not the dialogue is *transient*.

The *colour picker block* specifies the initial settings for the dialogue. This block is also used by other colour picker SWIs. Its format is as follows:

Offset	Meaning
0	flags:
	Bit **Meaning**
	0 1 ⇒ dialogue has a **None** button
	1 1 ⇒ dialogue has the **None** button selected
	2 - 3 dialogue *button type*, defining when Message_ColourPickerColourChanged is issued for it: 0 ⇒ never issued; 1 ⇒ issued on any change, except during drags, which give a message at drag end; 2 ⇒ issued on any change, including during drags
	4 1 ⇒ dialogue ignores Message_HelpRequest messages
	5 1 ⇒ dialogue does not pass on unhandled key presses to the Wimp (RISC OS 3.6 onwards)
4	pointer to the title to be used, or 0 for a default title
8	x coordinate of top left of the visible area of the dialogue
12	reserved (must be &80000000)
16	reserved (must be &7FFFFFFF)
20	y coordinate of top left of the visible area of the dialogue
24	reserved (must be 0)
28	reserved (must be 0)
32	colour descriptor (see page 5a-558)

Bits 0 and 1 of the flags control whether a **None** button appears between the colour patch and the **Cancel** button, and whether it is initially selected.

If the dialogue ignores Message_HelpRequest messages (page 3-242) directed to it, the client may send the Wimp message Message_HelpReply (page 3-243) to respond with its own text, or it may pass the message to the SWI ColourPicker_HelpReply (page 5a-572) to force the colour picker to reply with its own text. The client can hence replace the colour picker’s help text for some or all parts of the dialogue.

If bit 4 is set, it is up to the calling application – rather than the dialogue – to pass on unhandled key presses to the Wimp. This bit is ignored under RISC OS 3.5.

The returned dialogue handle is used as an argument to the other ColourPicker SWIs, and also in the Wimp messages that the ColourPicker module sends to the application to provide feedback on the user's selection of a colour.

Related SWIs

ColourPicker_CloseDialogue (page 5a-566)

Related vectors

None

ColourPicker_CloseDialogue (SWI &47703)

Closes a colour picker dialogue which is in progress

On entry

R0 = flags: all bits reserved (must be set to 0)
R1 = dialogue handle

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

This call closes a colour picker dialogue which is in progress. This may either be done by the client, or by the Wimp if the dialogue was set to be *transient* when created (see *Transient dialogues* on page 5a-557).

It is normally called in response to Message_ColourPickerCloseDialogueRequest (see page 5a-576).

Related SWIs

ColourPicker_OpenDialogue (page 5a-563)

Related vectors

None

ColourPicker_UpdateDialogue (SWI &47704)

Updates some or all of the contents of a colour picker dialogue

On entry

R0 = flags:

Bit	Part of dialogue to update when set
0	whether the dialogue has a **None** button
1	whether the dialogue has the **None** button selected
2	the button type of the dialogue
3	the visible area of the dialogue (RISC OS 3.6 onwards)
4	reserved (must be clear)
5	the window title
6	colour setting, from the colour descriptor's initial RGB triplet only
7	the colour model and setting, from the colour descriptor's model dependent data only (including optional data)
8	whether the dialogue ignores Message_HelpRequest events
9	whether the dialogue passes on unhandled key presses to the Wimp (RISC OS 3.6 onwards)

all other bits reserved (must be set to 0)

R1 = dialogue handle
R2 = pointer to a colour picker block (see page 5a-564)

On exit

R0 - R2 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call updates some or all of the contents of the colour picker dialogue whose handle is given. Only the parts of the box indicated by the flags word are updated.

The new values are taken from the passed colour picker block; all other parts of the block are ignored. The block need only be large enough to hold the highest offset field required.

The **None** button may be added or removed, and its setting adjusted. The title, setting and colour model may also be adjusted independently of each other.

If you are changing the visible area (ie bit 3 is set), you must fill in offsets 8 - 20 inclusive of the colour picker block (see page 5a-564), including the reserved words. This is for future compatibility.

If bit 7 is set then bit 6 is ignored; the colour model is updated, and the RGB triplet is calculated from the data in the colour model block. If bit 6 is set and bit 7 is clear, then the colour setting is updated from the colour descriptor’s initial RGB triplet and the colour model left unchanged, even if the current colour model isn't that in the colour descriptor.

Related SWIs

ColourPicker_OpenDialogue (page 5a-563)

Related vectors

None

ColourPicker_ReadDialogue (SWI &47705)

Reads the current state of a colour picker dialogue without changing it

On entry

R0 = flags: all bits reserved (must be set to 0)
R1 = dialogue handle
R2 = pointer to a buffer to hold a colour picker block (see page 5a-564),
or 0 to read required size

On exit

R0 preserved
R1 = window handle
R2 = required size of buffer (if 0 on entry); else preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads the current state of a colour picker dialogue without changing it. The state is returned as a colour picker block in the given buffer, which is assumed to be large enough to hold it. Because the size of the block may change when the colour model changes, you should always call this SWI twice: once to read the required size, then again to read the state of the colour picker dialogue.

Related SWIs

None

Related vectors

None

ColourPicker_SetColour (SWI &47706)

This call is reserved for future expansion; you must not use it in your own code.

ColourPicker_HelpReply (SWI &47707)

Makes the colour picker respond to a Message_HelpRequest with its own help text

On entry

R0 = flags: all bits reserved (must be zero)
R1 = pointer to Message_HelpRequest message block (page 3-242), as returned from Wimp_Poll (page 3-112)

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call makes the colour picker respond to a Message_HelpRequest with its own help text. It is typically used by a client that wishes to replace part of the colour picker's help system.

Such a client would, on creating and opening a colour picker dialogue, force it to ignore help requests (see page 5a-563). The client would then selectively respond to help requests itself, or use this call to make the colour picker respond instead.

Related SWIs

ColourPicker_OpenDialogue (page 5a-563)

Related vectors

None

ColourPicker_ModelSWI (SWI &47708)

This call is for internal use only; you must not use it in your own code.

Messages

Colour picker messages

These Wimp messages are generated by the Colour Picker in response to Wimp events on the colour picker dialogue. For more details about Wimp messages, see *Wimp_SendMessage* on page 3-193, and *Messages* on page 3-228.

Message_ColourPickerColourChoice (&47700)

This message is issued to the client whenever the user makes a definite choice of colour, by clicking Select or Adjust on the **OK** button of the dialogue. The format of the message block is:

R1+20	dialogue handle
R1+24	flags: bit 0 set ⇒ **None** chosen
R1+28…	colour descriptor chosen (see page 5a-558)

The colour descriptor gives the state of the dialogue so that a sensible default may be given next time the dialogue is used. When flags bit 0 is set, signifying that **None** was chosen, the colour descriptor will still be present.

The message may or may not be followed by a Message_ColourPickerCloseDialogueRequest.

Message_ColourPickerColourChanged (&47701)

This optional message is issued to the client when the colour displayed in the dialogue changes.

The format of the message block is:

R1+20	dialogue handle
R1+24	flags: bit 0 set ⇒ **None** chosen bit 1 set ⇒ drag in progress
R1+28…	colour descriptor chosen (see page 5a-558)

The client specifies when this message is to be sent (if at all) using *button type* flags, passed to ColourPicker_OpenDialogue (page 5a-563) as it creates and opens the dialogue.

Message_ColourPickerCloseDialogueRequest (&47702)

This message is issued to the client when the user dismisses the dialogue by clicking Select on the **OK** or **Cancel** icons or by using the Close icon of a toolbox dialogue. The client should respond by calling ColourPicker_CloseDialogue (page 5a-566) with the given handle. The format of the message block is:

R1+20	dialogue handle

This message is not issued for transient dialogues (see *Transient dialogues* on page 5a-557); the ColourPicker will instead automatically close the dialogue itself.

Message_ColourPickerOpenParentRequest (&47703)

This message is issued when the user opens a toolbox dialogue's parent by clicking Adjust or Shift-Adjust on its Close icon. If the colour picker has a parent dialogue box, the client should ensure that window is open and brought to the front. The format of the message block is:

R1+20	dialogue handle

In the former case (ie Adjust), the message will be followed by a Message_ColourPickerCloseDialogueRequest.

Message_ColourPickerResetColourRequest (&47704)

This message is issued to the client whenever the user clicks Adjust on the **Cancel** button of the dialogue. The format of the message block is:

R1+20	dialogue handle

The client should respond by calling ColourPicker_UpdateDialogue (page 5a-567), to reset the dialogue so that it displays the colour currently in effect. This is the last colour selected by clicking Adjust on the dialogue's **OK** icon, or failing that, the colour the dialogue showed when first opened.

This message is not issued under RISC OS 3.5.

* Commands

*ModelList

Lists all the loaded colour models

Syntax

`*ModelList`

Parameters

None

Use

*ModelList lists all the loaded colour models.

Example

```
*ModelList
0:      RGB
        The "physicist's model:" the quantity of each primary colour (red, green, blue).
1:      CMYK
        The "printer's model:" the quantity of each secondary colour (cyan, magenta, yellow), alongwith the key (black).
2:      HSV
        The "artist's model:" hue (or "tint"), saturation (or "shade") and value (or "tone").
```

Related commands

None

Related vectors

None

**ModelList*

130 Printing

Introduction

The printing system has been extended under RISC OS 3.5, mainly to support the vastly greater number of colours available. A new version of !Printers (1.22) has also been released for use under RISC OS 3, incorporating all changes relevant to the older hardware and operating system. Further changes have been made in RISC OS 3.6, largely to support printing JPEGs.

This chapter describes the few resultant changes the above have made in the programmer's interface. It does not describe in detail the rather more considerable internal changes made to the printing system.

Overview

Trapping of output calls

The printing system still works in the same way as before, intercepting the same calls. Support has been added to track extensions made in RISC OS, such as the use of mode specifiers, new sprite formats and wide translation tables. Because of this, the information in *Trapping of output calls* on page 3-573 is still correct; calls documented there as being processed by the printer drivers are still correctly handled.

There are some new calls handled by the printer drivers; these are documented below.

SpriteV

Treatment of SpriteOp reason codes

The table on page 3-586 shows the printer driver's treatment of each SpriteOp reason code in RISC OS 3. The table below shows the same information for the new reason code added in RISC OS 3.6:

Reason code	Meaning	Printer driver's treatment
17	Check the validity of a sprite area	Passed on

JPEG SWIs

RISC OS 3.6 provides calls to output JPEG images. The printer driver interacts with these calls using a mechanism broadly similar to that used for font output. When printing starts, the printer driver issues the SWI JPEG_PDriverIntercept (page 5a-161). The JPEG code then alters its SWI handling so that:

- It processes certain SWIs itself, as normal.
- It passes certain SWIs to the printer driver using the SWI PDriver_JPEGSWI (page 5a-587). The printer driver may then:
  1 process the call
  2 fault the call.
  The JPEG code does not process such SWIs itself.

The table below shows how each SWI is handled:

JPEG SWI	Meaning	Processing
Info	Gives information on a JPEG image held in a buffer	Processed by JPEG code as usual
File Info	Gives information on a JPEG image held in a file	Processed by JPEG code as usual
Plot Scaled	Decompresses, scales, and plots on the screen a JPEG image held in a buffer	Passed to printer driver and processed by it
Plot File Scaled	Decompresses, scales, and plots on the screen a JPEG image held in a file	Passed to printer driver and faulted by it: file operations are not allowed when printing
Plot Transformed	Decompresses, transforms, and plots on the screen a JPEG image held in a buffer	Passed to printer driver and processed by it
Plot File Transformed	Decompresses, transforms, and plots on the screen a JPEG image held in a file	Passed to printer driver and faulted by it: file operations are not allowed when printing
PDriver Intercept	Requests that SpriteExtend passes on all calls to JPEG plotting SWIs	Processed by JPEG code as usual

Technical details

The structure of the printing system

Front and back ends merged

In RISC OS 3.6, the front and back ends of the Printers application have been merged. It is important for future compatibility that any software you write does not assume the internal structure of the Printers application, as it may change again in subsequent releases of RISC OS.

New printer dumper

In RISC OS 3.6, a new printer dumper has been added for printers using Epson's Esc/P2 control language. The dumper's name is PDumperE2; its number is 6.

Colour input

The range of colour documents 'understood' as inputs remains unchanged on RISC OS 3 systems, save that PDriverDP has been extended to take in 8bpp, full palette sprites.

Support for new types of colour document has been added to the printing system as it has been added to other parts of RISC OS. So from RISC OS 3.5 onwards, 16/32 bpp true colour sprites are handled. From RISC OS 3.6 onwards, new type sprites of up to 8bpp with a palette and JPEG images – both files, and inserts in Draw files – are also handled.

New strip types

In order to support the improved colour output facilities some extra strip types have been defined from RISC OS 3.5 onwards, and all calls that use strip types have been extended to support them. The table below should replace that on page 3-675:

Value	Meaning	
0	monochrome	
1	grey scale	
2	256 colour	
3	Multiple pass 24 bit colour	(RISC OS 3 only)
4	Single pass 16 bit colour	(RISC OS 3.5 or later)
5	Single pass 24 bit colour	(RISC OS 3.5 or later)

Multiple pass 24 bit colour allows true colour output under RISC OS 3. Colour output is limited to 24bpp, and caches some very small colour conversion tables for optimum performance.

Single pass 16 and 24 bit colour strips only work under RISC OS 3.5 or later. Using a single pass gives faster output than using multiple passes. 16 bit colour renders internally using 16 bits of information; this is slightly faster than 24 bits, and requires less memory – but the images produced may contain slightly less colour information, depending on how the printer output palette has been defined. Attempting to use these strip types under RISC OS 3 will cause error messages from modules such as ColourTrans and from OS_SpriteOp calls; however, !Printers does not allow this, and so this is not a problem in normal use.

Pre-scanning of rectangles

From RISC OS 3.6 onwards the printer driver may choose to make a pre-scanning pass of the print rectangles that the application wants printed, provided the application is aware this may happen. Under RISC OS 3.6, when the bit image drivers are plotting a JPEG image they **must** do a pre-scan pass to ascertain memory requirements. In future, pre-scanning may be used for other purposes.

Whether or not the printer driver chooses to perform a pre-scan pass should be transparent to your application, which need only respond to all returned plotting rectangles as normal. A pre-scan pass should not increase printing times significantly provided that the majority of work your application does for each rectangle is to make plotting calls, which are simply ‘swallowed’ during the pre-scan.

Your application should not rely on information such as ColourTrans tables remaining valid between the pre-scan pass and the real pass.

New SWIs

From RISC OS 3.5 onwards you can enumerate the available strip types by calling PDriver_MiscOp with the new reason code of &80000002; for details see page 5a-586.

From RISC OS 3.6 onwards, there is a new SWI PDriver_JPEGSWI used to pass on JPEG SWIs to the printer drivers. For details, see page 5a-587.

Changes to existing SWIs

PDriver_Info (page 3-611)

A new bit has been added in RISC OS 3.6 to the features word returned in R3 that describes the printer driver:

Bit(s)	Value	Meaning
13	0	it does not expect a flag byte to be passed in R0 for PDriver_DrawPage (see below for details).
	1	it expects a flag byte to be passed in R0 for PDriver_DrawPage (see below for details).

PDriver_DrawPage (page 3-635)

The meaning of R0 has been altered in RISC OS 3.6; the top byte now holds flags. Currently only a single bit is used, to support pre-scanning. On entry:

R0 = number of copies to print, and flags:
bits 0 - 23 = number of copies to print
bit 24 set ⇒ application knows about pre-scan of rectangles by printer driver
bits 25 - 31 reserved (must be zero)

and on exit:

R0 = zero if finished; else more rectangles to be printed, and:
bit 24 set ⇒ this rectangle is for pre-scan only

We recommend that before calling this SWI you should first call PDriver_Info (see above) to check if the printer driver expects to receive the flag byte:

- If it does, you should set any relevant flags when calling this SWI (ie set bit 24), even if you believe you don't need to for the particular document you are printing. This ensures future compatibility.
- If it does not, you must not set any of the flags when calling this SWI, otherwise you will get a very large number of copies output!

PDriver_GetRectangle (page 3-637)

The meaning of R0 on exit has been extended in RISC OS 3.6 in the same way as for PDriver_DrawPage:

R0 = zero if finished; else more rectangles to be printed, and:
bit 24 set ⇒ this rectangle is for pre-scan only

PDumperReason_SetDriver (page 3-678)

Extra bits have been added to the configuration word:

Bit	Meaning when set	
4	Printer does black removal (PDumperLJ)	– RISC OS 3.5 onwards
5	Printer supports colour (PDumperLJ)	– RISC OS 3.6 onwards

PDumperReason_AbortJob (page 3-684)

This call has been extended from RISC OS 3.5 onwards so that the printer dumper can reset the printer should a print job be terminated (eg by Escape). R3 on entry has an extra bit flag from RISC OS 3.5 onwards, which if set also causes R4 to be used:

R3 bit 24 set ⇒ reset printer
R4 = pointer to copy of PDriverDP and dumper configuration data (see page 3-678)
– only if bit 24 of R3 is set

If a printer dumper is called with this bit set it must output to file the appropriate graphics termination and reset sequences to ensure the printer is in a sensible state.

This avoids problems where a job cancelled in the middle of a graphics sequence might leave the printer awaiting the rest of the sequence: the printer then hangs, and possibly even wrecks the next print job by treating its start as the end of the missing sequence.

PostScript restriction on 16 or 32bpp sprites with a mask

When the PostScript printer driver plots a 16 or 32bpp sprite with a mask to a colour printer, the masked pixels are white (not transparent), and so overwrite any graphics underneath the sprite. This is because the PostScript imaging model does not directly support a bitmap mask; the method used to emulate it for sprites with up to 8 bits per pixel does not generalise to sprites with a large number of colours.

This limitation only applies when output is to a colour printer; on black-and-white printers the sprite is reduced to 256 shades of grey, and the mask is transparent.

New palette file format in PDumperSupport

PDumperSupport has been enhanced in RISC OS 3.5 to allow better control over the quality of the printed image. To do so it uses a new format of palette file. In order to minimise prompting for the disc, these palette files are loaded into RMA and made available through ResourceFS. You can modify a palette file using the !RePRO tool, available from Oak Solutions.

SWI Calls

PDriver_MiscOp (SWI &8015A)

Returns a bit mask showing which strip types a printer dumper supports

On entry

R0 = &80000002 (reason code)
R1 = number of printer dumper for which to obtain strip types bitmask

On exit

R0 = supported strip types bit mask (if dumper loaded), otherwise preserved

Use

This call returns a bit mask showing which strip types a printer dumper supports (see page 5a-582). If bit *n* of the mask is set, then it shows that the printer dumper can output strip type *n*.

The new RISC OS 3 dumpers described in this chapter return 2_1111, and the RISC OS 3.5 dumpers return 2_110111. Older dumpers return 2_111.

PDriver_JPEGSWI (SWI &8015D)

Passes JPEG SWIs to the printer driver

On entry

R8 = JPEG SWI number modulo 64
All other registers as for original JPEG_… call

On exit

R8 corrupted (JPEG SWI's original R8 preserved by SpriteExtend)
All other registers preserved

Interrupts

Interrupt state is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call passes JPEG SWIs to the printer driver, once the printer driver has called JPEG_PDriverIntercept (page 5a-161) to enable interception. See *JPEG SWIs* on page 5a-580.

This call is for internal use only; you must not use it in your own code.

Related SWIs

JPEG_PDriverIntercept (page 5a-161)

Related vectors

None

131 Internationalisation

Introduction and Overview

This chapter describes some minor changes to the various internationalisation modules, all made in RISC OS 3.5.

International module

All calls (page 3-771 onwards)

The International module has been enhanced so that from RISC OS 3.5 onwards the strings it returns are terminated.

New countries

The following extra countries are supported from RISC OS 3.5 onwards:

Finland, Denmark, Austria, Belgium, Japan, MiddleEast, Netherlands, Switzerland, Wales.

Territory manager

Territory_Register (page 3-801)

From RISC OS 3.5, on entry to a territory module's SWI handler R0 is now always set to the current territory number. (If the SWI was called with a territory number of –1 to indicate the current territory, the territory manager resolves this before calling the SWI handler.)

On exit R0 should be preserved unless it is explicitly used to return a value from the SWI.

Territory_ConvertTimeToOrdinals (page 3-823)
Territory_ConvertTimeStringToOrdinals (page 3-825)
Territory_ConvertOrdinalsToTime (page 3-827)
Territory_SelectKeyboardHandler (page 3-831)
Territory_ReadCalendarInformation (page 3-847)

From RISC OS 3.5 onwards, these calls return the current territory in R0 on exit.

New MessageTrans SWI

A new SWI was added to MessageTrans in RISC OS 3.5:

- MessageTrans_Dictionary returns a pointer to the kernel's MessageTrans dictionary. This call is for internal use only; see page 5a-591.

SWI calls

MessageTrans_Dictionary (SWI &41509)

Returns a pointer to the kernel's MessageTrans dictionary

On entry

—

On exit

R0 = pointer to the kernel's MessageTrans dictionary

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns a pointer to the kernel's MessageTrans dictionary. (This is not the same as the OS_PrettyPrint dictionary, as described on page 1-536.)

Since the contents of this dictionary are liable to change with each successive release of RISC OS, this call is for internal use only; you must not use it in your own code.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

Part 18 – Miscellaneous

132 Sound

Introduction and Overview

From RISC OS 3.6 onwards the SoundDMA module has been extended to support 16 bit sound, as well as the 8 bit μ-law sound used by all earlier versions of RISC OS. This module is also supplied with the 16 bit Audio Card upgrade for the Risc PC.

Hardware

Machines using the new architecture all output sound using the VIDC20 video controller – whether it is a separate chip, or integrated into the ARM 7500 chip. VIDC20 provides two different types of sound output:

- It provides 8 bit μ-law sound on 8 channels; this is fully backward compatible with the sound provided by VIDC1 under earlier versions of RISC OS.

  This is output as an analogue signal, via internal sound DACs (digital-to-analogue converters) – just as with VIDC1.
- It also provides 16 bit linear stereo sound (ie CD-style).

  This is output as digital data, and requires an external DAC. The 16 bit Audio Card upgrade for the Risc PC provides such a DAC; one is fitted as standard to later designs of computer.

You may only use one of these types of sound output at a time; when 16 bit sound is fitted, the circuitry for 8 bit sound is disconnected or absent.

Configuration

The type of hardware fitted is set in CMOS RAM (see *CMOS RAM allocation* on page 5a-73)using the new command *Configure SoundSystem (page 5a-615). The configured sound hardware **must** match the actual hardware present, or you will get no sound output, and may get other unpredictable results. This configuration is read only when the SoundDMA module is initialised; hence you cannot adjust the hardware configuration without also re-initialising the SoundDMA module.

Under RISC OS 3.5, the default configuration is for an 8 bit sound system (ie no external 16 bit DAC is fitted). Under RISC OS 3.6 the kernel can detect the presence of 16 bit sound hardware and sets the default configuration accordingly: for a 16 bit sound system if it detects the hardware, and for an 8 bit sound system otherwise.

Technical details

The way the new SoundDMA module works depends on which type of VIDC20 sound output it is configured to use: 8 bit or 16 bit. You can determine how the sound system is configured by calling the new SWI Sound_Mode 0 (page 5a-603).

8 bit sound output

When the new SoundDMA module uses VIDC20's 8 bit sound output, it works just as on earlier versions of RISC OS. The standard VIDC20 internal μ-law DACs are used, as with VIDC1 and earlier versions of RISC OS.

16 bit sound output

The new features of the SoundDMA module become available when it is using VIDC20's 16 bit sound output.

16 bit sound is generated by a *linear handler*, which places 16 bit linear stereo sound samples in the sound DMA buffer. The SoundDMA module is responsible for requesting the linear handler to fill the buffer, and for outputting the data from this buffer to the external 16 bit DAC.

As before, the maximum size of the sound DMA buffer is one page, which is 4 Kbytes under RISC OS 3.6. Thus the maximum number of 16 bit stereo samples in the buffer is 1K, since each stereo pair takes up 32 bits (ie 4 bytes).

Sample rates

Sample rates for the 16 bit sound system are set and stored using a sample rate index:

- The sample rate index is an integer in the range 1 - *nsr* (the **n**umber of **s**ample **r**ates). As the index increases, so does the corresponding sample rate.

  You should not assume any particular value for *nsr*, nor any particular binding of index values to sample rates. Both may be affected by the sound hardware's configuration, and by future hardware or software developments.

For maximum portability and future compatibility, you should always fully determine the sample rates available from the sound system before using it:

- You can find the value of *nsr* by calling Sound_SampleRate 0 (page 5a-611).
- Once you have done this, you can then use Sound_SampleRate 2 (page 5a-613) to enumerate the available sample rates, or to find a match for a required sample rate.

Other reason codes for Sound_SampleRate allow you to read and set the sample index, and hence the sample rate:

- You can read the current sample rate index, and the corresponding sample rate, by calling Sound_SampleRate 1 (page 5a-612).
- You can set the current sample rate index by calling Sound_SampleRate 3 (page 5a-614).

Oversampling

Any digital sound system can generate an undesirable high-pitched noise that is correlated with the main signal; this is a by-product of the digital-to-analogue conversion process, and is more audible at lower sample rates. This high frequency *image* of the output analogue signal is sometimes inaccurately called the *alias*.

This effect can be reduced by a technique called *oversampling*. Extra samples are added between existing samples (typically by interpolation); the new sample is then played back at a higher rate, thus making the image noise less audible. A side effect can be a slight reduction in amplitude of higher frequencies; however in most cases this slight loss of ‘treble’ is outweighed in subjective terms by the benefit of reduced image noise level

You can make the RISC OS 16 bit sound system automatically use oversampling at all sample rates up to and including 25kHz:

- You can configure your preference using *Configure SoundSystem (page 5a-615)
- Applications can enable or disable oversampling, overriding the configured preference, by calling the new SWI Sound_Mode 1 (page 5a-604).

The output data stream is oversampled by a factor of two, by simple linear interpolation, before it reaches the DACs. This consumes a small amount of processor time on each sound system interrupt: at worst approximately 3% of a 30 MHz ARM610 processor with a selected sample rate of 25 kHz.

Note that when you are using oversampling the maximum number of samples a linear handler can place in the sound DMA buffer is halved to 512, so there is room for the extra interpolated samples.

Support for 8 bit sound

The new SoundDMA module also supports 8 bit µ-law sound. The sound is generated in the same way as before: the 8 bit Channel Handler generates µ-law data which it places in the sound DMA buffer. (For full details see *The Sound system* on page 4-3.) The SoundDMA module converts this data from multiple channels in 8 bit µ-law format to two stereo channels in 16 bit linear format. It then calls the linear handler (if any) to fill the DMA buffer with its own sound data; the linear handler can either overwrite the converted 8 bit sound data already in the buffer, or can merge it with its own sound data. All linear handlers should allow the user to configure their preference for this.

Restrictions of the 8 bit emulation

This conversion of 8 bit sound to 16 bit sound is transparent, and in general no difference from the old 8 bit sound system will be apparent. However:

- Although the 16 bit sound system provides many of the sample periods possible under the old 8 bit sound system – including the default period of 48µs used by the standard voice generators and many applications – there are a few less commonly used periods for which it can only provide a close match.
- The 8 bit samples must fit within the sound DMA buffer when they are converted to 16 bit stereo sound, and hence must total less than a page in size. This means that the number of 8 bit samples is limited to 1K without oversampling, or 512 with oversampling. Again, this is not a problem with the 8 bit default of 208 samples per channel.

Changes to existing SWIs

Sound_Configure (page 4-18)

When the computer is configured for 16 bit sound, this call affects both 16 bit sound and the emulated 8 bit sound:

- R0 gives the number of channels for 8 bit sound, just the same as ever. It is ignored by the 16 bit sound system, which always has two channels (the left and right stereo pair).
- R1 was originally defined as the ‘sample length (in bytes per channel)’. You should now think of R1 as giving ‘the number of samples per channel’ for both 8 and 16 bit sound. The two definitions are effectively the same for 8 bit sound, but the new definition also covers 16 bit sound.
- R2 sets the sample period for both 8 bit and 16 bit sound. However, the new SWI Sound_SampleRate (page 5a-609) is the preferred way to control sample rates for 16 bit sound.

When you have 16 bit sound configured, the values of R0 and R1 must be such that 8 bit, converted 8 bit and 16 bit sound data can all fit within a page, which is the maximum size of the sound DMA buffer. If not, the number of samples is set to the highest value for which all three types of data will fit. Also, not all sample periods can be provided; in such cases the sample period is set to the closest match. As always, you can check the number of samples and sample period actually set by calling Sound_Configure with null parameters.

Linear handlers

Registering linear handlers

A linear handler registers itself with SoundDMA by calling the new SWI Sound_LinearHandler 1 (page 5a-605). When registering, you give the address of the handler code – which is called to fill the sound DMA buffer – and a parameter passed to the handler in R0. Typically the parameter will be a pointer to a data area containing any information the handler may need to perform its task. The address and parameter of the previous linear handler (if any) are returned.

Only one linear handler can be registered with the SoundDMA module. You should therefore only register your linear handler immediately before starting to play sound, and should re-register the previous handler as soon as you have finished.

You can find which linear handler is currently registered by calling Sound_LinearHandler 0 (page 5a-607).

How linear handlers are called

The handler is passed the address of the sound DMA buffer for it to fill with 16 bit linear stereo sound data. Each sample is stored in a word as a pair of signed (2's complement) 16 bit values, with the right channel data in bits 0 - 15, and the left channel data in bits 16 - 31. A flag indicates if the buffer already contains sound data converted from multiple channels in 8 bit μ-law format; see *Support for 8 bit sound* on page 5a-598 for more details of the action the linear handler should take in this case.

The full conditions for entry and exit are as follows:

On entry

R0 = parameter passed in R2 to Sound_LinearHandler 1 when registering
R1 = pointer to quadword-aligned sound DMA buffer
R2 = pointer to word immediately after sound DMA buffer
R3 = flags:

bits 0 - 2		initial buffer content indicator:
	0	data in buffer is invalid and must be overwritten
	1	data in buffer has been converted from multiple channels in 8 bit μ-law format, and is not all 0.
	2	data in buffer is all 0: if handler would generate silent output, it may simply return.
	3 - 7	reserved
bits 8 - 31		reserved, and should be ignored

R4 = sample rate for playback, measured in units of $1/1024$ Hz; for example 20 kHz (20000 Hz) would be passed as 20000×1024, which is 20480000

On exit

R0 - R10 may be corrupted
R11, R12, R13 must be preserved

Interrupts

Interrupts may be enabled during execution of the handler

Processor mode

Handler may be called in either IRQ mode or SVC mode
Processor mode must be preserved on exit

SWI calls

Sound_Mode (SWI &40144)

Examines and controls the 16 bit sound system's configuration

On entry

R0 = reason code
Other registers depend on reason code

On exit

Registers depend on reason code

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call examines and controls the 16 bit sound system's configuration.

The particular action of Sound_Mode is given by the reason code in R0 as follows:

R0	Action	Page
0	Reads the current sound system configuration	5a-603
1	Enables or disables automatic oversampling	5a-604

Related SWIs

None

Related vectors

None

Sound_Mode 0 (SWI &40144)

Reads the current sound system configuration

On entry

R0 = 0 (reason code)

On exit

R0 = sound system capabilities:
- 0 ⇒ the sound system only supports 8 bit μ-law sound; R1 is 0, and R2 preserved
- 1 ⇒ the sound system supports 16 bit sound, and also 8 bit μ-law sound by emulation; other registers as below

R1 = the configuration stored in SoundSystem bits at offset 132 of CMOS RAM (see *CMOS RAM allocation* on page 5a-73):

bits 0 - 1	16 bit sound control configuration, from bits 5 - 6
bits 2 - 3	reserved
bit 4	16 bit sound quality configuration, from bit 7
bits 5 - 31	reserved

Use

This call reads the current sound system configuration. Any new sound applications you write – particularly those capable of 16 bit sound output – should always call this SWI to determine whether the configured sound output hardware supports 16 bit sound output.

- If the configured hardware does not support 16 bit sound output, R0 is 0 on return. You should only use the original sound system SWIs – in particular Sound_Configure – to determine and control sound output parameters such as sampling rate. Other Sound_Mode reason codes are not available, nor are SWIs in the range &40145 - &4017F inclusive. The sound system will behave in a fully backward compatible manner.
- If the configured hardware does support 16 bit sound output, R0 is 1 on return. You can use all Sound_Mode reason codes, and the Sound_LinearHandler and Sound_SampleRate SWIs. R1 gives an indication of any external sound clock hardware facilities present, and the configured state of automatic oversampling. A subset of the original sound system's sample rates are available; see *Restrictions of the 8 bit emulation* on page 5a-598.

Sound_Mode 1 (SWI &40144)

Enables or disables automatic oversampling

On entry

R0 = 1 (reason code)
R1 = new state of automatic linear 2× oversampling: 0 ⇒ disabled, 1 ⇒ enabled

On exit

R0 preserved
R1 = previous state of automatic linear 2× oversampling: 0 ⇒ disabled,
1 ⇒ enabled

Use

This call enables or disables automatic linear 2× oversampling, overriding the default set in CMOS RAM by *Configure SoundSystem (page 5a-615).

For a description of oversampling, see *Oversampling* on page 5a-597.

Sound_LinearHandler (SWI &40145)

Examines and controls the 16 bit linear stereo sound handler

On entry

R0 = reason code
Other registers depend on reason code

On exit

R0 preserved
Other registers depend on reason code

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call examines and controls the 16 bit linear stereo sound handler.

The particular action of Sound_LinearHandler is given by the reason code in R0 as follows:

R0	Action	Page
0	Returns the current 16 bit linear stereo sound handler	5a-607
1	Registers or removes the 16 bit linear stereo sound handler	5a-608

You must not use this call unless 16 bit sound hardware is configured, as determined by a preceding call of Sound_Mode 0 (see page 5a-603).

Related SWIs

None

Related vectors

None

Sound_LinearHandler 0 (SWI &40145)

Returns the current 16 bit linear stereo sound handler

On entry

R0 = 0 (reason code)

On exit

R0 preserved
R1 = pointer to current handler code, or 0 if no handler is installed
R2 = parameter passed in R0 to current handler, or –1 if no handler is installed

Use

This call returns the current 16 bit linear stereo sound handler, giving the address of the handler code, and the parameter passed to it in R0.

Sound_LinearHandler 1 (SWI &40145)

Registers or removes the 16 bit linear stereo sound handler

On entry

R0 = 1 (reason code)
R1 = pointer to new handler code, or 0 to remove the handler
R2 = parameter passed in R0 to handler, or –1 if removing the handler

On exit

R0 preserved
R1 = pointer to previous handler code, or 0 if no handler was installed
R2 = parameter passed in R0 to previous handler, or –1 if no handler was installed

Use

This call registers or removes the 16 bit linear stereo sound handler. When registering, you give the address of the handler code – which is called to fill the sound DMA buffer – and a parameter passed to the handler in R0. The address and parameter of the previous linear handler (if any) are returned.

Only one linear handler can be registered with the SoundDMA module. You should therefore only register your linear handler immediately before starting to play sound, and should re-register the previous handler as soon as you have finished.

Sound_SampleRate (SWI &40146)

Determine/control sound sample rate

On entry

R0 = reason code
Other registers depend on reason code

On exit

R0 preserved
Other registers depend on reason code

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call controls the sound sample rate.

The particular action of Sound_SampleRate is given by the reason code in R0 as follows:

R0	Meaning	Page
0	Reads the number of available sample rates	5a-611
1	Reads the current sample rate index, and the corresponding sample rate	5a-612
2	Reads the sample rate corresponding to a sample rate index	5a-613
3	Sets the current sample rate index	5a-614

You must not use this call unless 16 bit sound hardware is configured, as determined by a preceding call of Sound_Mode 0 (see page 5a-603).

Related SWIs

None

Related vectors

None

Sound_SampleRate 0 (SWI &40146)

Reads the number of available sample rates

On entry

R0 = 0 (reason code)

On exit

R0 preserved
R1 = number of available sample rates, or *nsr* (see page 5a-596)

Use

This call reads the number of available sample rates, or *nsr*.

You need to know this value to ensure that the sample rate index you must pass to most other Sound_SampleRate reason codes is in the required range 1 - *nsr* (see *Sample rates* on page 5a-596).

Sound_SampleRate 1 (SWI &40146)

Reads the current sample rate index, and the corresponding sample rate

On entry

R0 = 1 (reason code)

On exit

R0 preserved
R1 = current sample rate index, in the range 1 - *nsr* (see page 5a-596)
R2 = current sample rate, in units of $1/1024$ Hz

Use

This call reads the current sample rate index, and the corresponding sample rate, measured in units of $1/1024$ Hz. For example a sample rate of 20 kHz (20000 Hz) would be returned in R2 as 20000×1024, which is 20480000.

Sound_SampleRate 2 (SWI &40146)

Reads the sample rate corresponding to a sample rate index

On entry

R0 = 2 (reason code)
R1 = sample rate index to be read, in the range 1 - *nsr* (see page 5a-596)

On exit

R0, R1 preserved
R2 = sample rate corresponding to the given sample rate index, in units of $1/1024$ Hz

Use

This call reads the sample rate corresponding to a sample rate index, in units of $1/1024$ Hz. For example a sample rate of 20 kHz (20000 Hz) would be returned in R2 as 20000×1024, which is 20480000.

Once you have called Sound_SampleRate 0 to find the number of available sample rates (*nsr*), you can then:

- Enumerate the available sample rates by repeatedly making this call with R1 set to all valid indexes (ie 1 - *nsr* inclusive).
- Find a particular sample rate (or the closest approximation, if acceptable) by using this call in a 'binary chop' algorithm, since sample rates increase monotonically with increasing sample rate index.

Sound_SampleRate 3 (SWI &40146)

Sets the current sample rate index

On entry

R0 = 3 (reason code)
R1 = new sample rate index, in the range 1 - *nsr* (see page 5a-596)

On exit

R0 preserved
R1 = previous sample rate index
R2 = previous sample rate, in units of $1/1024$ Hz

Use

This call sets the current sample rate index.

It returns the previous sample rate index, and the corresponding sample rate measured in units of $1/1024$ Hz. For example a sample rate of 20 kHz (20000 Hz) would be returned in R2 as 20000×1024, which is 20480000.

* Commands

*Configure SoundSystem

Sets the configured value for the type of sound hardware to use

Syntax

```
*Configure SoundSystem 8bit | 16bit [oversampled] | n
```

Parameters

`8bit`	standard 8 bit µ-law sound, as on older hardware
`16bit`	standard 16 bit sound, as on newer hardware or Acorn 16 bit Audio Card
`oversampled`	perform sample interpolation to keep sample rate over 24kHz
n	value 0 - 7 to store in SoundSystem bits of CMOS RAM (at offset 132, bits 5 - 7: see *CMOS RAM allocation* on page 5a-73)

Use

*Configure SoundSystem sets the configured value for the type of sound hardware to use, and whether to use oversampling for 16 bit sound.

For a description of oversampling, see *Oversampling* on page 5a-597.

Example

```
*Configure SoundSystem 16bit oversampled
```

Related commands

None

133 CompressJPEG

Introduction and Overview

The CompressJPEG module is available from RISC OS 3.6 onwards. It provides SWIs with which you can compress raw image data into a JPEG image. It is a port of release 5 of the Independent JPEG Group's software.

The module is not in the RISC OS 3.6 ROM, but is instead held in the System application. If you wish to use the module in a program, you should first use the following command to ensure it is loaded:

```
RMEnsure CompressJPEG 0.00 RMLoad System:Modules.JCompMod
```

To compress raw image data into a JPEG image, you start by calling CompressJPEG_Start (page 5a-619), which sets up the compression environment. You then compress each row of the source image with a separate call to CompressJPEG_WriteLine (page 5a-621). Finally you finish the compression by calling CompressJPEG_Finish (page 5a-622).

Technical details

How JPEG images are compressed

JPEG files encode colour pictures as YUV (Y = intensity, U and V are colour) data. Compressing involves the following steps:

- Convert RGB data to YUV.
- Throw away 3 out of 4 of the U and V pixels.
- Convert 8×8 tiles of Y, U and V values through a Discrete Cosine Transform, into an 8×8 square of frequency coefficients.
- Discard coefficients which are zero, or close to zero. This will tend to change the visual appearance of the picture very little.
- Reduce the accuracy with which the remaining coefficients are held (known as ‘quantisation’). Again, this changes the appearance very little. The amount by which this is done, controls the compression factor of the image. By now, most of the coefficients will be zero.
- Reorder the 64 coefficients in a zig-zag order, which increases the average length of runs of zeros in the coefficient block.
- Huffman-encode the resulting stream of values.

(Incidentally, decompression involves reversing these steps.)

SWI calls

CompressJPEG_Start (SWI &4A500)

Starts the JPEG compression process, setting up various parameters for it

On entry

R0 = pointer to buffer for JPEG data
R1 = size of JPEG data buffer
R2 = pointer to block of parameters:

+0	width of image in pixels
+4	height of image in pixels
+8	quality value (0 - 100): lower quality results in a smaller image
+12	number of 8 bit components in source: 3 ⇒ 24 bit colour, 1 ⇒ 8 bit greyscale
+16	horizontal DPI of image, or 0 if unknown
+20	vertical DPI of image, or 0 if unknown

R3 = pointer to workspace area, or 0 for the CompressJPEG module to allocate its own workspace from the RMA
R4 = size of workspace area (if R3 ≠ 0)

On exit

R0 = JPEG tag, to be passed to other CompressJPEG SWIs

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call starts the JPEG compression process, setting up various parameters for it.

The buffer for the JPEG data should be as large as possible, since the JPEG compression routines cannot guarantee to compress the image by a fixed amount.

If you wish to supply your own workspace area, its required size for a colour (24 bit) image is:

20000 + ((image width rounded up to a multiple of 16) × 30)

and its required size for a greyscale (8bit) image is:

20000 + ((image width rounded up to a multiple of 16) × 9)

An error is returned if the workspace area becomes full.

Related SWIs

CompressJPEG_WriteLine (page 5a-621), CompressJPEG_Finish (page 5a-622)

Related vectors

None

CompressJPEG_WriteLine (SWI &4A501)

Compresses one row of source pixels into the JPEG buffer

On entry

R0 = JPEG tag
R1 = pointer to a row of pixels:

For colour:	a buffer of continuous RGB values – ie a byte stream of the format R, G, B, R, G, B…
For greyscale:	a buffer of continuous 8 bit gray values

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call compresses one row of source pixels into the JPEG buffer. It should be called once for each row of the source data.

An error is returned if the JPEG buffer becomes full.

Related SWIs

CompressJPEG_Start (page 5a-619), CompressJPEG_Finish (page 5a-622)

Related vectors

None

CompressJPEG_Finish (SWI &4A502)

Finishes the JPEG compression process, returning the size of the complete image

On entry

R0 = JPEG tag

On exit

R0 = size of JPEG image within the buffer

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call finishes the JPEG compression process, returning the size of the complete image. Any workspace claimed by the CompressJPEG module for the compression is released.

Related SWIs

CompressJPEG_Start (page 5a-619), CompressJPEG_WriteLine (page 5a-621)

Related vectors

None

Example program

The pseudo-code below shows you how you might convert a 32 bpp sprite into a JPEG:

```
/* Pseudo C code for converting a 32bpp sprite to a JPEG */

Allocate buffer for JPEG = JPEG_buffer;
Allocate buffer for workspace = workspace_buffer;
Allocate buffer for line of source pixels = line_buffer;

argument_block arguments;
arguments.width = sprite_width_in_pixels;
arguments.height = sprite_height_in_pixels;
arguments.quality = quality;
arguments.components = 3;
arguments.horizontal_dpi = 0;
arguments.vertical_dpi = 0;

sprite_pointer = start_of_data_within_sprite;

JPEG_tag = CompressJPEG_Start(JPEG_buffer, JPEG_buffer_size, arguments,
        workspace_buffer, workspace_buffer_size);

for loop = 1 to sprite_height_in_pixels {
        convert_sprite_data_to_rgb(sprite_pointer, line_buffer);
        CompressJPEG_WriteLine(JPEG_tag, line_buffer);
        sprite_pointer += sprite_width_in_words;
}
CompressJPEG_Finish(JPEG_tag);
```

134 Expansion card support

Introduction and Overview

The expansion card interface has been enhanced in several ways for RISC OS 3.5. It now supports:

- 32 bit wide data paths
- a directly mapped area of 16MB per card, known as EASI space
- an interface dedicated to a network card
- Direct Memory Addressing (DMA).

This chapter covers the changes that have been made in order to support these enhancements; all these changes apply from RISC OS 3.5onwards.

Technical details

Using EASI space

EASI space is an extension of the existing space giving a directly mapped area of 16MB for each expansion card. The address of this space is set in the RISC OS ROM.

ROMs in EASI space

The expansion card bus is electrically capable of having ROMs (or EPROMs) connected, which RISC OS can then read. The ROMs are only 8 bits wide and are copied once at start up into RAM.

Under earlier versions of RISC OS, this was always done using a loader and paging register. However, from RISC OS 3.5 onwards there is no need for this if a ROM is mapped into EASI space, since RISC OS does the loading itself. Mapping a ROM into EASI space has other advantages: access to the entire ROM address space is faster, and not having loaders frees-up ROM space.

The format for a ROM in EASI space is the same as that for a ROM in the normal expansion card space; it must contain the same ECId information. However, since the size restriction is lifted there is no need to have a second Chunk directory accessed through the loader. Note that although the ROM is in the EASI space, the interrupt relocations are still relative to the base of expansion card space.

Determining where a ROM is to be loaded

RISC OS will cope with a ROM in only one of expansion card space and EASI space, not both at once. When determining which is present, it first checks for a ROM in expansion card space by reading location 0. If bit 1 is low it assumes there is a ROM in expansion card space, and does not access EASI space.

If you wish to use a ROM in EASI space, it is vital that your expansion card either does not respond to reads of location 0, or provides data with bit 1 set high. Failure to do this will make RISC OS read spurious data as it attempts to load a non-existent ROM from expansion card space, and ignores the ROM in EASI space.

It also follows that you must not map read-sensitive hardware into location 0, or its state may be altered as RISC OS attempts to load ROMs at boot time.

Finding EASI space

You can read the logical and physical addresses of the area and its size by calling Podule_ReadInfo (page 5a-631). The returned addresses are stable as long as the machine configuration is stable, and therefore only need be read once, after a reset.

The network card

ROMs on the network card

RISC OS 3.5 – and later versions – loads the ROM on a network card itself, in a similar manner to ROMs mapped into EASI space. For this loader to work, it is vital that your network card conforms to the current hardware specification.

The format for a ROM on a network card is also the same as that for a ROM in the normal expansion card space. It must contain the same ECId information; the interrupt relocations must be present and all be set to zero. Since the loader is effectively loaded before the enumeration begins, there is again no need to have a second Chunk directory.

SWIs and the network card

Most SWIs work with the network card, simply by quoting its ROM section when calling (see *ROM sections* on page 5a-628). You should note the following:

- Podule_ReadBytes (page 4-145) reads the network ROM image using the loader built in to RISC OS.
- Podule_WriteBytes (page 4-147) will not accept the network ROM section, because its ROM space is treated as read only.
- Podule_CallLoader (page 4-149) will not accept the network ROM section, because the loader isn't valid.
- Podule_RawRead (page 4-151) and Podule_RawWrite (page 4-153) access the network card’s device address space.
- Calls such as Podule_HardwareAddress (page 4-155) and Podule_HardwareAddresses (page 4-159) return the device address.

ROM sections

New ROM section numbers

ROM section numbers have been allocated for a further four expansion cards, and for the network card. The network card is the highest numbered one, and is last in the printout from *Podules.

The new numbers are:

ROM section	Meaning
4	Expansion card 4
5	Expansion card 5
6	Expansion card 6
7	Expansion card 7
8	Network card

New ways of specifying the ROM section

All expansion card SWIs (with the single exception of Podule_ReturnNumber) use R3 to specify which expansion card or extension ROM to access. Some calls can access both, and are documented as accepting a ROM section number; others can access only expansion cards, and are documented as accepting an expansion card slot number (ie a subset of ROM sections).

As well as ROM section numbers, these SWIs now also accept a hardware base address (as returned by Podule_HardwareAddress or Podule_HardwareAddress), whether or not it is combined with a CMOS address.

The ‘formal definition’ of what is acceptable in R3 is as follows (demonstrated by the following pseudo code):

```
CASE
WHEN Value = -1: System ROM ==> Error "System ROM not acceptable as
                                 Expansion Card or Extension ROM number"
WHEN Value <= -2 AND >= -16: Extension ROM(-Value-1)
WHEN Value >= 0 AND <= 31: Expansion Card(Value)
WHEN Value AND &FFE73000 = &03240000: Expansion Card((Value AND &C000)>>14)
WHEN Value AND &FFE73000 = &03270000: Expansion Card(4+(Value AND &C000)>>14)
WHEN Value AND &FFFF3FFF = &03000000: Expansion Card((Value AND &C000)>>14)
WHEN Value AND &FFFF3FFF = &03030000: Expansion Card(4+(Value AND &C000)>>14)
WHEN Value >= &70 AND <=&7F: Expansion Card((Value AND &C)>>2)
WHEN Value >= &3C AND <=&4F: Expansion Card(7-((Value AND &C)>>2))
WHEN Value = EASILogicalBase(0..7): Expansion Card(0..7)
WHEN Value = EASIPhysicalBase(0..7): Expansion Card(0..7)
OTHERWISE Error "Bad Expansion Card or Extension ROM number"
ENDCASE
```

Changes to the combined hardware address

The definition of the combined hardware address has had to be changed to allow for the introduction both of the network card and of processors with 32 bit addressing.

The combined hardware address consists of the base address of CMOS RAM and the base address of an expansion card or extension ROM, OR'd together. The bits that are set in one address can be guaranteed unused in the other, because the two addresses are so widely separated. By using two different masks, the two addresses can be extracted.

In earlier versions of RISC OS:

- All expansion cards had a base address above &1000, and so the lower 12 bits of the combined address were used for the CMOS base address.
- The processor used 26 bit addressing, so the top 6 bits of the combined address were unused

However, under RISC OS 3.5 and later:

- The network card has a base address below &1000, and so now only the lower 10 bits of the combined address are used for the CMOS base address (which is still sufficient).
- The processor supports 32 bit addressing, so the combined address now uses all 32 bits.

The new definition is thus:

Bits	Meaning
0 - 9	base address of CMOS RAM – expansion cards only (10 bits)
12 - 32	bits 12 - 32 of base address of expansion card/extension ROM

All this has really done is to move the boundary between the two parts of the combined address. Existing expansion cards and extension ROMs will continue to work, because their base address under RISC OS 3.5 will still only have bits 12 - 25 set, as before.

These changes apply both to SWIs returning combined hardware addresses, and to the entry points for loaders. Entry points in new expansion cards should now extract the hardware base address by masking the incoming register value thus:

```
LDR Rmv, =2_00000000000000000000001111111111
BIC Rba, Rha, Rmv
```

or thus:

```
LDR Rmv, =2_11111111111111111111110000000000
AND Rba, Rha, Rmv
```

and should obtain the CMOS base address thus:

```
LDR Rmv, =2_11111111111111111111110000000000
BIC Rca, Rha, Rmv
```

or thus:

```
LDR Rmv, =2_00000000000000000000001111111111
AND Rca, Rha, Rmv
```

Simple expansion card descriptions

Some expansion cards use only a simple ECId, where the product is identified by a 4 bit ID field unique to that product. However, there is no way of providing a textual description of the product. Support for this has been added from RISC OS 3.5 onwards.

The description is held in the file:

`Resources:$.Resources.Podule.Messages`

It is looked up as a token consisting of the string `Simple` followed by a single hexadecimal digit giving the ID field (which must be 1 - F). For example, the line that is looked up for an ID field of 1 is:

```
Simple1:Acorn Econet
```

This method is used to extend *Podules (to return a description of simple expansion cards. The description can also be read using Podule_ReadInfo.

New chunk type for device data

A new chunk type has been defined for device data (see *Operating System Identity Byte* on page 4-129). The value 9 indicates a two byte chunk used to store a CRC of the ROM, typically only used by proprietary diagnostic and test software.

SWI calls

Podule_ReadInfo (SWI &4028D)

This call returns a selection of data specific to a given expansion card

On entry

R0 = bitmask of required results (see below)
R1 = pointer to buffer to receive word aligned word results
R2 = length in bytes of buffer
R3 = ROM section (see page 4-133 and page 5a-628)

On exit

R0, R1 preserved
R2 = length of results
R3 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 returns a selection of data specific to the given expansion card. The information required is specified by bit flags. The data is returned in single words, which are placed into the user supplied buffer at word intervals, in the same order as the bit flags (ie data for the lowest bit set is at the lowest address).

The bit flags are:

Bit	Value to return when set
0	Expansion card/Extension ROM number
1	Normal (synchronous) base address of hardware
2	CMOS address
3	CMOS size in bytes
4	Extension ROM or network ROM base address
5	Expansion card ID
6	Expansion card product type
7	Combined hardware address
8	Pointer to description (zero for no description)
9	Address of EASI space
10	Size of the EASI space in bytes
11	Logical number of the primary DMA channel
12	Logical number of the secondary DMA channel
13	Address of Interrupt Status Register
14	Address of Interrupt Request Register
15	Address of Interrupt Mask Register
16	Interrupt Mask value
17	Device Vector number (for IRQ)
18	Address of FIQ as Interrupt Status Register
19	Address of FIQ as Interrupt Request Register
20	Address of FIQ as Interrupt Mask Register
21	FIQ as Interrupt Mask value
22	Device Vector number (for FIQ as IRQ)
23	Address of Fast Interrupt Status Register
24	Address of Fast Interrupt Request Register
25	Address of Fast Interrupt Mask Register
26	Fast Interrupt Mask value
27	Ethernet address (low 32 bits)
28	Ethernet address (high 16 bits)
29	Address of MEMC space (zero for no space)
30 - 31	Reserved (must be zero) – error if set

The description strings may be in temporary buffers (for example, MessageTrans error buffers) so it is wise to copy them to private workspace before calling any other SWIs.

When updating any of the nine interrupt registers it is essential that both IRQ and FIQ are disabled for the duration.

This SWI supersedes other expansion card SWIs such as Podule_HardwareAddress.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

Podule_ReadID (page 4-139), Podule_ReadHeader (page 4-140), Podule_HardwareAddress (page 4-155), Podule_HardwareAddresses (page 4-159), Podule_ReturnNumber (page 4-161)

Related vectors

None

Podule_SetSpeed (SWI &4028E)

Changes the speed of access to expansion card hardware

On entry

R0 = new speed required:
0 ⇒ No change, 1 ⇒ IOMD+ timing type A, 2 ⇒ IOMD+ timing type B
3 ⇒ IOMD+ timing type C, 4 ⇒ IOMD+ timing type D
R3 = ROM section (see page 4-133 and page 5a-628)

On exit

R0 = previous speed setting
R3 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 changes the speed of access to expansion card hardware. The kernel initialises all expansion cards' access speed to type A.

This call is only available from RISC OS 3.5 onwards.

Related SWIs

None

Related vectors

None

Application Notes

Reading the machine supplied value for the Ethernet address should ideally be carried out using the following code. Note that this is not the only way to get the required result, but it is the recommended way:

```
GetEthernetAddress

; Entry;
;       R3 ==> Any recognisable part of podule addressing
;
; Exit;
;       R0 ==> Low 32 bits of the Ethernet address
;       R1 ==> High 16 bits of the Ethernet address

        STMFD   sp!, { r0-r2, r4, lr }
        MOV     r0, &18000000           ; Bits for read high and low
        MOV     r1, sp                  ; Point to the buffer
        MOV     r2, #8                  ; Size of buffer
        SWI     XPodule_ReadInfo
        LDMVCFD sp!, { r0-r2, r4, pc }  ; Return with results if OK
        MOV     r4, r0                  ; Save the original error
        MOV     r0, #0                  ; Start at the first chunk
Loop
        SWI     XPodule_EnumerateChunks
        BVS     ErrorExit
        TEQ     r0, #0
        BEQ     ErrorExit               ; End of list, so not found
        TEQ     r2, #&F7                ; Ethernet Address?
        BNE     Loop
        TEQ     r1, #6                  ; Wrong size is a failure
        BNE     ErrorExit
        SUB     r0, r0, #1              ; Back to the chunk we liked
        MOV     r2, sp                  ; Pass in the data pointer
        SWI     XPodule_ReadChunk
        LDMVCFD sp!, { r0-r2, r4, pc }  ; Return with results if OK
ErrorExit
        CMP     pc, #&80000000          ; Set V
        STR     r4, [ sp, #0 ]          ; Original error Podule_ReadInfo
        LDMFD   sp!, { r0-r2, r4, pc }
```

135 Joystick module

Introduction and Overview

The Joystick module has been extended in RISC OS 3.6 to provide support for PC-style analogue joysticks, as well as the Atari-style digital joysticks supported by earlier versions of RISC OS.

Support has also been added for calls used with analogue input devices on older Acorn machines.

Technical details

Changes to existing SWIs

Joystick_Read (page 4-218)

Joystick_Read has been extended to support reason codes. In RISC OS 3.6 these are used to specify the format in which to return the read values: 8 bit, or 16 bit (available for analogue only). For full details, see page 5a-647.

New SWIs

Joystick calibration

Different analogue joysticks will output different voltages when in the same position. Two new SWIs have been added to calibrate the voltages, so that all analogue joysticks return consistent values when their position is read. These are:

- Joystick_CalibrateTopRight (page 5a-651)
- Joystick_CalibrateBottomLeft (page 5a-651)

OS_Byte calls

You should also see *OS_Byte calls* on page 5a-639 for details of OS_Byte calls added.

Acorn I/O expansion card compatibility

Previously, analogue input devices could be connected to a RISC OS computer using the Acorn I/O Podule's ADC port.

Pinout of connectors

The old I/O Podule and the new joystick interface use the same type of connector. However, the pinout used by a PC-style joystick – and hence by the new joystick interface – differs from that used by the I/O Podule's ADC port. You will therefore need an adaptor cable to connect devices intended for the old I/O Podule to the new joystick interface.

Backward compatibility of software

The I/O Podule provides various OS_Byte calls and the BASIC ADVAL command to support its ADC port. If there is no I/O Podule present then the Joystick module provides the same calls, which instead access the joystick interface (provided it has been configured for analogue input).

OS_Byte calls

The OS_Byte calls provided are:

- OS_Byte 16 (page 5a-640), which stores the number of channels to be sampled
- OS_Byte 17 (page 5a-641), which returns to the caller, doing nothing (rather than forcing an ADC conversion, as on the I/O Podule)
- OS_Byte 128, 0-4 (page 5a-642), which returns the switch state and last channel converted, or a channel's uncalibrated position
- OS_Byte 188 (page 5a-644), which reads the current channel
- OS_Byte 189 (page 5a-645), which reads the number of channels to be sampled
- OS_Byte 190 (page 5a-646), which reads the resolution of conversion.

Differences between the I/O Podule's hardware and the joystick interface's hardware mean that not all the OS_Byte calls provide identical functionality in both implementations. However, the vast majority of I/O Podule software should still run using the joystick interface, without change.

The BASIC ADVAL keyword

ADVAL is a BASIC function that takes a single parameter. The Joystick module adds support for parameters 0 - 4:

- ADVAL (0) returns an integer giving the state of switch 0 on joysticks 0 (in bit 0) and 1 (in bit 1).
- ADVAL (1) returns an integer giving the raw position of channel 1; this is uncalibrated, in the range 0 - 65535.

  Similarly, ADVAL (2), (3) and (4) return respectively the raw position of channel 2, 3 and 4.

All other ADVAL parameters continue to work in the same way as always; they are documented in the *BBC BASIC Reference Manual*.

SWI calls

OS_Byte 16 (SWI &06)

Stores the number of channels to be sampled

On entry

R0 = 16 (reason code)
R1 = number of channels to be sampled (0 - 4)

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 stores the number of channels to be sampled. If the value passed is greater than 4, it is stored as 4. This stored value can be read using OS_Byte 189.

(On the I/O Podule this call also set the number of channels to be sampled; but this is not possible on the joystick interface's hardware.)

Related SWIs

OS_Byte 189 (page 5a-645)

Related vectors

None

OS_Byte 17
(SWI &06)

Returns to the caller, doing nothing

On entry

R0 = 17 (reason code)
R1 = channel number on which to force ADC conversion (0 - 4) – not implemented

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 returns to the caller, doing nothing.

(On the I/O Podule this call forced an ADC conversion on the given channel; but this is not possible on the joystick interface's hardware.)

Related SWIs

None

Related vectors

None

OS_Byte 128, 0-4 (SWI &06)

Returns the switch state and last channel converted, or a channel's uncalibrated position

On entry

R0 = 128 (reason code)
R1 = sub-reason code:

0	return switch state and number of last channel converted
1 - 4	channel number for which to return position

On exit

R0 preserved
R1 = state of switch 0 on joysticks 0 (in bit 0) and 1 (in bit 1) – if R1 = 1 on entry;
or low byte of 16 bit uncalibrated position for channel given in R1 on entry
R2 = number of last channel converted – if R1 = 1 on entry;
or high byte of 16 bit uncalibrated position for channel given in R1 on entry

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 switch state and last channel converted, or a channel's uncalibrated position, depending on the value passed in R4:

- If R4 is zero on entry, this call returns the state of one switch on each of the first two joysticks; the reason only two values are returned is for backward compatibility with the I/O Podule software. This call also returns the number of the last channel used for ADC conversion.

- If R4 is a channel number on entry (ie 1 - 4), this call instead returns the uncalibrated position of that channel, in the range 0 - 65535.

(On the I/O Podule this call does the same.)

For details of other OS_Byte 128 sub-reason codes, see page 1-170.

Related SWIs

None

Related vectors

None

OS_Byte 188 (SWI &06)

Reads the current channel

On entry

R0 = 188 (reason code)

On exit

R0 preserved
R1 = current channel (1 - 4)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call reads the current channel.

(On the I/O Podule this call does the same.)

Related SWIs

None

Related vectors

None

OS_Byte 189 (SWI &06)

Reads the number of channels to be sampled

On entry

R0 = 189 (reason code)

On exit

R0 preserved
R1 = number of channels to be sampled (0 - 4)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call reads the number of channels to be sampled, as stored using OS_Byte 16.

(On the I/O Podule this call does the same.)

Related SWIs

OS_Byte 16 (page 5a-640)

Related vectors

None

OS_Byte 190 (SWI &06)

Reads the resolution of conversion

On entry

R0 = 190 (reason code)

On exit

R0 preserved
R1 = 0 (default conversion, which is 16 bit)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call reads the resolution of conversion. This is always returned as 0, meaning the default for the hardware, which is 16 bit.

(On the I/O Podule this call can return 0 for the default (which for its hardware is 12 bit), or 8 for 8 bit conversion, or 12 for 12 bit conversion; however, hardware limitations mean that conversion can only be guaranteed to 8 bits.)

Related SWIs

None

Related vectors

None

Joystick_Read (SWI &43F40)

Returns the state of a joystick

On entry

R0 = joystick number and reason code:

bits 0 - 7	joystick number (0 ⇒ first joystick, 1 ⇒ second, etc)
bits 8 - 15	reason code
bits 16 - 31	reserved (must be zero)

On exit

Registers depend on reason code

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI is used to obtain the state of the requested joystick. The format in which the state is returned is set by the reason code in bits 8 - 15 of R0:

Reason	Meaning	Page
0	Returns the 8 bit state of a digital or analogue joystick	5a-649
1	Returns the 16 bit state of an analogue joystick	5a-650

Related SWIs

None

Related vectors

None

Joystick_Read 0 (SWI &43F40)

Returns the 8 bit state of a digital or analogue joystick

On entry

R0 = joystick number and reason code:

bits 0 - 7	joystick number (0 ⇒ first joystick, 1 ⇒ second, etc)
bits 8 - 15	0 (reason code)
bits 16 - 31	reserved (must be zero)

On exit

R0 = 8 bit joystick state:

bits 0 - 7	signed Y value in the range –127 (down) to 127 (up); for a single switch joystick, –64 ⇒ down, 0 ⇒ rest, and 64 ⇒ up
bits 8 - 15	signed X value in the range –127 (left) to 127 (right); for a single switch joystick, –64 ⇒ left, 0 ⇒ rest, and 64 ⇒ right
bits 16 - 23	switches (eg fire buttons) starting in bit 16; unimplemented switches return 0
bits 24 - 31	reserved

Use

This reason code returns the 8 bit state of a digital or analogue joystick.

For an analogue joystick, this call reads the last conversion made; it does not force a conversion itself. Furthermore, conversions are not started until you first call this SWI. That first call always returns X = 0, Y = 0, and no switches closed, since there is no completed conversion to read.

Applications which are only interested in state (up, down, left, right) should not simply test the bytes for positive, negative or zero. We recommend that the ‘at rest’ state should span a middle range, say from –32 to 32, since you cannot always rely upon analogue joysticks to produce a particular value when at rest.

This reason code is available from RISC OS 3 onwards. (In earlier versions of the *RISC OS 3 Programmer's Reference Manual* it was referred to simply as Joystick_Read, since reason codes were not in use.)

Joystick_Read 1 (SWI &43F40)

Returns the 16 bit state of an analogue joystick

On entry

R0 = joystick number and reason code:

bits 0 - 7	joystick number (0 ⇒ first joystick, 1 ⇒ second, etc)
bits 8 - 15	1 (reason code)
bits 16 - 31	reserved (must be zero)

On exit

R0 = 16 bit joystick position:

bits 0 - 7	signed Y value in the range 0 (down) to 65535 (up)
bits 8 - 15	signed X value in the range 0 (left) to 65535 (right)

R1 = joystick switch state:

bits 0 - 7	switches (eg fire buttons) starting in bit 0; unimplemented switches return 0
bits 8 - 31	reserved

Use

This reason code returns the 16 bit state of an analogue joystick.

For an analogue joystick, this call reads the last conversion made; it does not force a conversion itself. Furthermore, conversions are not started until you first call this SWI. That first call always returns X = 0, Y = 0, and no switches closed, since there is no completed conversion to read.

Applications which are only interested in state (up, down, left, right) should not simply test the bytes for minimum, middle, and maximum values. We recommend that the ‘at rest’ state should span a middle range, say from 24576 (&6000) to 40960 (&A000), since you cannot always rely upon analogue joysticks to produce a particular value when at rest.

This reason code is available from RISC OS 3.6 onwards.

Joystick_CalibrateTopRight (SWI &43F41)

Calibrates analogue joysticks to return the full range of values

On entry

—

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 calibrates analogue joysticks to return the full range of values. You should make this call with all joysticks held in the top right position.

To calibrate, you must call both this SWI and Joystick_CalibrateBottomLeft. Once you have called one of this pair of SWIs, Joystick_Read (page 5a-647) and the ADVAL command return an error, until you have completed the process of calibration by calling the other one of the pair. The read calls will then return their full range of values.

Related SWIs

Joystick_CalibrateBottomLeft (page 5a-652)

Joystick_CalibrateBottomLeft (SWI &43F42)

Calibrates analogue joysticks to return the full range of values

On entry

—

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 calibrates analogue joysticks to return the full range of values. You should make this call with all joysticks held in the bottom left position.

To calibrate, you must call both this SWI and Joystick_CalibrateTopRight. Once you have called one of this pair of SWIs, Joystick_Read (page 5a-647) and the ADVAL command return an error, until you have completed the process of calibration by calling the other one of the pair. The read calls will then return their full range of values.

Related SWIs

Joystick_CalibrateTopRight (page 5a-651)

136 Monitor power saving

Introduction and Overview

Government agencies and independent organisations worldwide are involved in setting limits or goals for power consumption in office equipment, in order to slow the growth in overall demand for electric power. Desktop computers are one of these agencies' primary targets, especially their displays, which are a significant portion of their power consumption.

VESA (the Video Electronics Standards Association) has produced a proposed standard called Display Power Management Signalling (DPMS), which provides a common means for a display controller to send a signal to the display that makes it enter various power management states. DPMS is likely to be adopted by most major monitor manufacturers.

Where the monitor supports this mechanism, RISC OS 3.5 and later versions can use it. This has been done by incorporating DPMS into the code used in RISC OS 3 to blank the screen after the computer has been left untouched for a certain amount of time.

Technical Details

The DPMS power saving states are distinguished by the presence or absence of pulses on the horizontal and vertical sync lines.

State	Power saving	Recover time	Horiz. sync	Vert. sync	Video
On	None	None	Pulses	Pulses	Pulses
Stand by	Minimal	Short	No Pulses	Pulses	Blanked
Suspend	Substantial	Longer	Pulses	No Pulses	Blanked
Off	Maximum	System Dependent	No Pulses	No Pulses	Blanked

To be compliant with DPMS, displays do not necessarily have to have all four states, but they must implement at least one reduced power consumption state.

From RISC OS 3.5 onwards the screen blanking mechanism has been extended so that it can select any of the power states above:

- The existing RISC OS 3 screen blanking mechanism (ie blanking the video whilst leaving the sync pulses active) must still be possible. This is to avoid problems with older monitors which do not support DPMS and require the presence of sync pulses.
- Screen blanking must be able to select all three reduced power states, since the DPMS proposed standard does not specify which of the three states a DPMS monitor must support.

Controlling DPMS power saving states

The DPMS power saving state to enter during screen blanking for a particular monitor is configured by an optional line in its ModeInfo file (see *ModeInfo files* on page 5a-106). The different states are specified using the values 0 - 3:

Value	Meaning
0	DPMS disabled - screen blank just blanks video.
1	Screen blank enters 'Stand-by' mode.
2	Screen blank enters 'Suspend' mode.
3	Screen blank enters 'Off' mode.

There is no simple user interface to set or alter this value.

The power saving state is actually set by a video control parameter; see *Service_ModeExtension* on page 5a-125. The control index is 11, and the value is in the range 0 to 3, with the same meanings as above. The line in the ModeInfo file makes the ScreenModes module append such an entry to all VIDC lists it passes to the kernel.

137 The Toolbox modules

Introduction and Overview

The *Acorn C/C++* product introduced the RISC OS Toolbox, which makes it much easier to write consistent, high-quality desktop applications whose user interface complies with the RISC OS 3 Style Guide. The key parts of the Toolbox are:

- A number of *object modules*, each of which provides code that handles an *object* (ie a part of the user interface of a desktop application such as a window, or a menu, or an icon on the icon bar) and the *components* that make up that object (such as menu entries, buttons and sliders)
- ResEdit, which is an interactive editor for designing the different objects in the application's user interface, and saving them to a *resource file*
- ResTest, which is an application to check the appearance and behaviour of all the objects in a resource file
- The Toolbox module itself, which is at the core of the system; it provides a layer of abstraction between an application and the Wimp, loads objects from resource files, and calls the code in object modules
- The TinyStubs module, which provides TinySupport_… SWIs for internal use within the Toolbox.

Advantages of the Toolbox

Using the Toolbox has a number of advantages. In particular:

- The object modules provide much of the code needed to handle your user interface, so you don't need to write the code yourself
- ResEdit and ResTest provide a much quicker and easier way of designing user interfaces than the past method, which involved designing window templates and creating other components of your user interface (such as menus) in your application's code.
- The Toolbox modules support multiple applications, so their code can be shared, avoiding unnecessary duplication of code, and hence cutting down on memory usage.

Toolbox modules in RISC OS

To cut down still further on the memory requirement of applications written to use the Toolbox, the RISC OS 3.6 ROM contains all its modules (ie the Toolbox module itself, the TinyStubs module, and each of the object modules). Toolbox applications therefore don't need to load the modules into RAM, and much of their user interface is implemented by shared code that runs from ROM. The object modules supplied are:

Module	Provides
ColourMenu	a menu for selecting a desktop colour
ColourDbox	a dialogue box for selecting any colour
DCS	a dialogue box for discard/cancel/save for unsaved data, and a dialogue box for handling quit with unsaved data
FileInfo	a dialogue box showing information on a given file
FontDbox	a dialogue box for selecting font characteristics
FontMenu	a menu for selecting a font
Iconbar	an icon on the left or right of the iconbar
Menu	a Wimp menu
PrintDbox	a dialogue box for selecting print options
ProgInfo	a dialogue box for showing program information
SaveAs	a dialogue box for saving data by icon drag
Scale	a dialogue box for selecting a scale factor
Window	a Wimp window

Toolbox documentation

The Toolbox is documented in the *User Interface Toolbox* guide, supplied with *Acorn C/C++*.

Writing applications to use the Toolbox

To write applications that use the Toolbox, you will need to purchase *Acorn C/C++*, so that you have:

- documentation
- the means to create resource files (ie ResEdit)
- a binary distribution licence for the Toolbox modules so you can supply them with your application, and it can hence run on RISC OS 3.1 and 3.5.

Applications communicate with the Toolbox using standard RISC OS mechanisms such as SWIs (known as *Toolbox methods*) and Wimp events (known as *Toolbox events*). You therefore don't have to write Toolbox applications using the languages supplied with Acorn C/C++ (ie C, C++, and ARM assembler); you can use other languages such as BASIC.

Appendixes

138 Appendix A: Warnings on the use of ARM assembler

Early versions of ARM 7 series processors corrupt the cache when code performs a store multiple to the last word in a cache line, which is in the cache, but is not written through the write buffer. These processors are fitted only to a **very** few Acorn computers.

To work round this problem, all areas of memory that can be cached must also use the write buffer. This requires that:

- All page tables that mark pages/sections as cacheable must also mark them as bufferable.
- The control register must never be set up such that the cache is on, but the write buffer is disabled.
- When the cache is disabled it is also flushed (as advised in the ARM710 datasheet).

You must ensure that your own code follows these guidelines.

RISC OS does not contravene these guidelines, except for versions of ROMPatch supplied with RISC OS 3.5, a fixed version of which has been supplied with the very few processor upgrades that may show this fault.

139 Appendix B: File formats

Draw files

The Draw file format (see page 4-463) has been extended in RISC OS 3.6:

Objects

A new object has been defined for including JPEG images within a Draw file. It uses the same object header as other Draw objects; see page 4-465. The rest of the data for the object is as follows:

JPEG object

Object type number 16

Size	Description
4	width of image, in Draw units
4	height of image, in Draw units
4	x pixel density, in dpi
4	y pixel density, in dpi
24	transformation matrix
4	length *n* of the JPEG image data
n	JPEG image data in JFIF format
0 - 3	up to 3 bytes, to pad to a word boundary

The first four words can be derived from information returned by JPEG_Info (page 5a-149). The transformation matrix is as described in Font_Paint (see page 3-437), in the same format used elsewhere in the Draw module and for other Draw file objects.

For more details of JPEG images, see *JPEG images* on page 5a-145, and *CompressJPEG* on page 5a-617.

The Draw applications supplied with RISC OS 2 and RISC OS 3 do not use this object type.

140 Appendix C: Errata and omissions for RISC OS 3 PRM

This appendix contains a number of errata and omissions for the *RISC OS 3 Programmer's Reference Manual*, together with clarification of some text.

Unless otherwise specified, the comments below for any given operation or call refer to all versions of RISC OS that support it.

InsV, RemV, CnpV (page 1-88)

The documentation for each of these vectors states that 'it must be called with interrupts disabled… therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.'

From RISC OS 3 onwards, the default owner of the vector is the buffer manager, which disables interrupts itself. Calling code need no longer disable interrupts, and code claiming the vector should no longer assume that interrupts are disabled on entry.

PaletteV (page 1-105)

An undocumented reason code was added in RISC OS 3; this is R4 = 6. The reason code is reserved for internal use.

Device numbers (page 1-120)

For models using the 82C710 or 82C711 peripheral controller (eg the A5000), device numbers 11 and 12 were transposed. They should read:

11 Floppy disc interrupt from 82C710/711
12 IDE hard disc interrupt

Events (page 1-147)

Events may not be received in the order in which they are generated.

Internet receive event and Internet transmission status event (page 1-161)

These events are not used by DCI4 versions of the Internet module, such as the one in RISC OS 3.6.

Code offset (page 1-217)

R1 is undefined on entry in the case of a configuration keyword.

Help and command keyword table (page 1-216)

If the byte 1 of the information word is such that the final parameter is GSTrans'd, the command tail passed to the module will have a trailing space.

SWI handler code (page 1-220)

The SWI handler code is **not** passed the value of R9 specified by the caller. The RISC OS SWI despatcher corrupts R9 before calling a module's SWI handler code, and on exit from the handler restores R9 to the value specified by the caller.

This may be fixed in future versions of RISC OS so that the SWI despatcher passes R9 uncorrupted to and from the SWI handler code. In preparation for this, you should ensure that any SWI handler code does not incorrectly corrupt R9. (Currently this would be hidden by the SWI despatcher preserving R9 around the SWI handler code.)

Also, the example code on page 1-221 is wrong; it makes a SWI call in SVC mode without preserving R14, and then uses the corrupted R14 to return. It should read:

```
.UnknownSWIError
        STMFD   R13!, {R14}                     ; Push R14 to call SWI in SVC
        ADR     R0, ErrToken
        MOV     R1, #0
        MOV     R2, #0
        ADR     R4, ModuleTitle                 ; From module header
        SWI     "XMessageTrans_ErrorLookup"
        LDMFD   R13!, {R14}                     ; Pull R14
        ORRS    PC, R14, #Overflow_Flag
```

OS_Module 20 (page 1-252)

This reason code is not available in RISC OS 2.

Unused SWI (page 1-295)

The first paragraph of this description should read:

This handler is called by the default owner of the UKSWIV. (When a SWI is called, RISC OS first checks if it is a kernel SWI; it then checks if it is a module SWI by looking at its hash table constructed from the headers of initialised modules. It then calls 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 – which is the kernel – calls the Unused SWI handler.)

The default handler returns the error ‘SWI &xxxxxxxx not known’, or just ‘SWI not known’ if the SWI was called from an IRQ process.

OS_ReadVarVal (page 1-314)

If you are checking for the existence/length of a variable (ie bit 31 of R2 is set on entry), R0 is corrupted on exit.

Transient CallBacks (page 1-296)

You must not rely on any relationship between the order in which Transient CallBacks are added and the order in which they are called.

Transient CallBacks are not called between successive lines of an Obey file, nor when screen scrolling is disabled by the *Scroll Lock* or *Ctrl-Shift* keys.

OS_RemoveTickerEvent (page 1-445)

You cannot use this call to remove a ticker event from within that event’s own code. Instead, your ticker event must call OS_AddCallBack (page 1-324) to add a transient CallBack that makes the call to OS_RemoveTickerEvent.

OS_CheckModeValid (page 1-742)

For all versions of RISC OS, this call returns –2 to indicate there is ‘insufficient memory’ if the **currently** allocated amount of screen memory is too little for the specified mode. It does not take into account whether the area could grow.

File operations (page 1-774)

You may get unpredictable results when using *ScreenLoad to load a sprite that was not created by *ScreenSave. The same applies to the equivalent SWIs.

Pixel translation table (page 1-780)

A number of calls that use a pixel translation table specify it as optional. You can only omit it if the sprite you are plotting has the same number of bits per pixel as the current screen mode. We recommend you always supply a table, and leave it to RISC OS to ignore it if it is unnecessary.

Creating sprites (page 1-775)

If you try to create a sprite with a palette, the palette is incorrect if it is for a different number of bits per pixel to that used by the current mode. The best workround is to create the sprite without a palette, and then to add the palette.

OS_SpriteOp (page 1-788)

OS_SpriteOp is not re-entrant.

OS_SpriteOp 60 (page 1-838)

The purpose of the save area is to preserve your own context should anyone switch output away from you.

OS_HeapSort (page 1-970)

In the section *Advanced features*, it states that bit 31 may optionally be used in conjunction with bit 30. In fact, setting bit 30 (ie build word-array of pointers pointed to by R1 from R4,R5) also causes the bit 31 action to be taken (ie sort true objects pointed to by R4 after sorting the pointers). Thus if you wish to sort only the pointers and not the records to which they point, you must build the pointer array yourself, rather than setting bit 30 to have this call build it.

IIC_Control (page 1-977)

R0 is corrupted on exit.

FileSwitch (page 2-11)

All calls that open a file for writing when it cannot be written to (eg write-protected media, no write access, locked filing system) do not generate an error. The error is not generated until an attempt is actually made to write to the file.

Special fields (page 2-14)

The root directory $ was omitted from the example, which should read:

```
 net#MJHardy::disc1.$.mike
    #MJHardy::disc1.$.mike
-net#MJHardy-:disc1.$.mike
   -#MJHardy-:disc1.$.mike
```

File$Path and Run$Path (page 2-18), Using other path variables (page 2-19)

When using path variables you must remember that they may specify multiple objects, and hence there are clear limitations. Reading an object specified by a correctly constructed path will always work; but writing or deleting objects using a path may be undefined in behaviour, and may hence be disallowed.

Filing system numbers (page 2-21)

The entry for DOSFS refers to a stand alone filing system called DOSFS, released with certain versions of the PC Emulator. All image filing systems (including the DOSFS supplied in RISC OS 3 onwards) use a filing system number of 0 to distinguish them from ordinary filing systems.

OS_FSControl 41 (page 2-126)
OS_FSControl 42 (page 2-127)

These calls return incorrect information for NetFS.

Disc formats (page 2-199)

The ‘perfect’ disc formats referred to in this section may not always be attainable. For example, the 710/711 controllers cannot achieve a gap1 of more than 255 bytes, and hence use a good alternative. See also FileCore_DiscFormat (page 2-236) and ADFS_VetFormat (page 2-291) for a description of the process used to negotiate an attainable format.

Entries (page 2-212)

The *NewDirAtts* are as follows:

Bit	Meaning when set
0	Object has owner read access
1	Object has owner write access
2	Object is locked
3	Object is a directory
4	Object has public read access
5	Object has public write access
6	Reserved (must be zero)
7	Reserved (must be zero)

FileCore_MiscOp (page 2-240)

The cross references to the various reason codes should read:

Value	Meaning	Page
0	Mount	2-242
1	Poll changed	2-244
2	Lock drive	2-246
3	Unlock drive	2-247
4	Poll period	2-248
5	Eject disc	2-249

Software protection schemes (page 2-267)

Limitations of disc controllers place further restrictions on using 128 byte sectors:

- Always create the master disc with a machine that has a 1772 disc controller
- Only read a single 128 byte sector at a time.

ADFS_SetIDEController (page 2-298)

In the versions of ADFS supplied before RISC OS 3.5, R4 must be 1 on entry (ie the interrupt status must be in bit 0).

DOSFS (page 2-323)

The mapping of DOS attributes to RISC OS attributes is not described in this chapter. It is as follows:

If a DOS file is read only, its RISC OS attributes are LWR; otherwise they are RW. If a RISC OS file is locked, it is a read only file when transferred to DOS; otherwise it is a read/write file.

Other attributes are preserved where possible, using a mechanism that is subject to change and so not documented.

Directory structure (page 2-416)

This section – and others in the chapter – describe the Apps and Fonts directories as containing 'ROM-resident' objects. This is not so for all versions of RISC OS; some or all of these objects may be on disc.

OS_SerialOp (page 2-468)

Two reason codes were added to OS_SerialOp in RISC OS 3, but were not documented in the *RISC OS 3 Programmers Reference Manual*. These are described later in this chapter:

- OS_SerialOp 7 (page 5a-690) is for internal use only.
- OS_SerialOp 8 (page 5a-691) reads/writes the serial input buffer threshold value. It is provided as a replacement for OS_Byte 203 (page 2-462), and you should use it in preference.

Redirection (page 2-496)
printer: (page 2-497)

When using *Copy to send a file to the `printer:` system device, you should ensure you are using the F copy option. For example:

```
*Copy myfile printer: ~CF~V
```

Free_Register (page 2-522)

The free space routine should exit using the instruction:

```
LDMIA   R13!,{PC}
```

FSEntry_Open and ImageEntry_Open (page 2-541)

The documentation states that – for FSEntry_Open – reason code 1 is only used by RISC OS 2. It can in fact be called by other versions of RISC OS under certain very specific conditions.

FSEntry_Func 33 (page 2-587)

The heading for this section should read 'FSEntry_Func 33 and ImageEntry_Func 33', since this entry point can be called for an image filing system – as stated in its description.

Descriptor block flags (page 2-598)

Under RISC OS 2, bit 2 when set means that the FileCore module supports background operations. The bit is reserved (as documented) only from RISC OS 3 onwards.

Wherever possible, you should make hard discs support mount like floppies do, and hence set bit 4. If you do not do so, FileCore may have trouble mounting discs that use an alien format, as it then has no way of determining their geometry, and so has to make some assumptions that may be invalid

Returning errors (page 2-603)

This section should read:

If there is no error then R0 must be zero on exit and the V flag clear. If there is an error then V must be set and R0 must be one of the following:

Value	Meaning
< &100	internal FileCore error number
Bit 30 set, bit 31 clear	pointer to error block
Bit 30 clear, bit 31 set	disc error bits: bits 0 - 20 = disc byte address / 256 bits 21 - 23 = drive bits 24 - 29 = disc error number

For a list of internal FileCore error numbers, see the section entitled *Returning errors* on page 2-600.

MiscOp entry (page 2-604)

When this entry point is called, R12 is a pointer to your FileCore module's private word. All other registers are as documented (ie the same values as were passed to FileCore_MiscOp). In general, all FileCore_MiscOp calls are passed straight through to your FileCore module, which should implement their full functionality; however, FileCore counts lock/unlock calls itself, and only calls your module when it should actually lock or unlock the drive.

Port numbers (page 2-649)

The following port numbers are also reserved:

Port	Allocation
&A0	SJ Research *FAST protocol (file server management)
&AF	SJ Research Nexus net finder reply port

Service_EconetDying (page 2-653)

When this service call is issued, the Econet module is already being finalised, and you may not make further calls to it. Resources such as ports, CBs etc are no longer valid, and you may dispose of any relevant local workspace.

Layout of windows (page 3-10)

The last line of page 3-13 should read:

work_area_pixel_at_origin_x = scroll_offset_x – visible_area_min_x
work_area_pixel_at_origin_y = scroll_offset_**y** – visible_area_max_y

Similarly, the 'entire formula' given near the top of the next page should read:

work area x = screen_x + (scroll_offset_x – visible_area_min_x)
work area y = screen_y + (scroll_offset_**y** – visible_area_max_y)

Misc icons (page 3-33)

There is no 'acorn' icon; the Task Manager uses the 'switcher' icon referred to in *Icon bar icons* on page 3-32.

Wimp_Initialise (page 3-85)

The description of R3 on entry is wrong, and should read:

R3 = pointer to a list of message numbers terminated by a 0 word (not if R0 is less than 300). If Wimp version number is ≥310 then a null pointer indicates that **no** messages are important to this task, whereas a null list indicates that **all** messages are important; this is the reverse of what you might expect.

Wimp_CreateIcon (page 3-93)

Icon validation strings are order dependent; they are scanned from left to right.

If you use 2 icons with the 'S' validation string they must both be the same size.

Wimp_Poll (page 3-112)

From RISC OS 3 onwards, on exit, if R0 is 18 (User_Message_Recorded) then R2 is set to the task handle of the sender.

Wimp_DecodeMenu (page 3-158)

The returned string is terminated.

Wimp_ReadPalette (page 3-189)

From RISC OS 3 onwards, if R2 is 'TRUE' on entry (ie &45555254), then the returned palette entries are 24 bit rather than 12 bit: ie &*bbggrrnn* rather than &*b*0*r*0*g*0*nn*. This saves having to copy the top nibbles into the bottom nibbles before making ColourTrans calls.

Wimp_SpriteOp (page 3-198)

R1 is corrupted on exit.

Message_RAMFetch (page 3-253)

The versions of !Edit supplied before RISC OS 3.5 only respond to this message if it is sent as a User_Message_Recorded (ie if acknowledgement is requested).

Message_WindowInfo (page 3-256)

The section heading for this message should read 'Message_WindowInfo…', not 'Message_WindowInf…'. The description of the message should read:

R1+20	window handle
R1+24	reserved (must be 0)
R1+28	*`string`* giving trailing part of sprite name to use, null terminated – sprite name used is `ic_`*`string`*
R1+36	string giving title to use, null terminated; this should be as short as possible, and may be truncated by the iconiser (eg Pinboard truncates at a space or at the 10th character, whichever is shorter)

TaskWindow_Input (page 3-263)

Location R1+24 of the message block holds the input data itself, not a pointer to it. The data needs no terminator, because its length is held in R1+20.

TaskWindow_Ego (page 3-263)
TaskWindow_Morio (page 3-263)

The versions of !Edit supplied before RISC OS 3.5 only respond to these messages if they are sent as a User_Message (ie if no acknowledgement is requested).

TaskWindow_NewTask (page 3-264)

The versions of !Edit supplied before RISC OS 3.5 only respond to this message if it is sent as a User_Message (ie if no acknowledgement is requested).

The command passed in this message is only the head of the command that must be issued via Wimp_StartTask. The full command is:

`command xxxxxxxx yyyyyyyy` *nb there is a trailing space!*

where *`xxxxxxxx`* and *`yyyyyyyy`* are the `task` and `txt` parameters passed when creating the task window (see **TaskWindow* on page 3-324).

Filter_RegisterPostFilter (page 3-306)

Under RISC OS 3, if a filter routine sets R0 to –1 to claim an event and prevent it being passed to its task, then that event is not passed on to any further post filters. From RISC OS 3.5 onwards, claiming an event does not prevent other post filters from being called, but does still prevent the event being passed to the task.

TaskWindow (page 3-319)

Changing screen mode from task windows can have unpredictable results.

*TaskWindow (page 3-324)

See *TaskWindow_NewTask* above for correct information on how to respond to this message.

ColourTrans_SelectTable (page 3-344)

The cross reference to ColourTrans_GenerateTable should refer to page 3-405, not page 3-346.

ColourTrans_SelectTable (page 3-344)
ColourTrans_GenerateTable (page 3-405)

If R0 is 256 on entry, it is assumed not to point to a sprite area, but R1 is still assumed to point to a sprite. This special value is useful if you need to use sprites that are not held in a sprite area. For example, Draw uses it for sprites that are held in a Draw file without a preceding sprite area control block.

Thus R0 is only assumed to be a pointer if it is greater than 256.

*FontInstall (page 3-522)

FontInstall will only rescan a directory already on the path if it moves to the head of the path. The best way to force a re-scan after changing a directory known to the Font Manager is to call *FontRemove, then *FontInstall.

*LoadFontCache (page 3-526)
*SaveFontCache (page 3-527)

A saved font cache is only valid if RMA usage is the same as when it was saved, since it contains absolute pointers to RISC OS modules and their workspace. If RMA usage has altered (eg the cache is loaded to a different address, or the Font Manager's workspace is in a different location) you will get no error on loading the cache; but you will get many subsequent errors. These calls are therefore deprecated.

Winding rules (page 3-536)

The first sentence of the description of the even-odd winding rule should read:

Even-odd means that an area is filled if a ray from that area to outside the path's bounding box crosses an odd number of paths.

Line thickness (page 3-541)

The second bullet point should read:

- If the thickness is *n*, then the line will be drawn with a thickness of *n*/2 user coordinates translated to pixels on either side of the theoretical line position.

DrawV when printing (page 3-581)

The rounding of coordinates is printer driver specific. Some drivers may not output paths that are less than one output device pixel wide. However, paths of width 0 (ie ‘as thin as possible’) should always result in output.

Service_PDriverChanged (page 3-610)

This service call is only issued when the PDriver sharer module has selected a new printer driver. This means it is not issued if the currently selected printer driver is deselected, but no new one is selected.

PDriver_SelectJob (page 3-622)

Under RISC OS 3.1 and earlier, R7 is corrupted on exit.

PDriver_Reset (page 3-630)

The state of the printer driver after this call is not necessarily the same as it is after initialisation. For example, the PostScript printer driver does not know of any fonts (see PDriver_MiscOp on page 3-656).

PDriver_SelectIllustration (page 3-644)

We now recommend that the user should explicitly choose when a print job is to be saved to file for use as an illustration in another document. Only if the user has made that choice should you call this SWI; you should call PDriver_SelectJob for all other printing.

PDriver_EnumerateDrivers (page 3-655)

The values on exit are:

R0 = handle to enumerate next driver, or zero if no more
R1 = printer driver number (page 3-604) if R0 ≠ 0, or undefined if no more

Printer definition files (page 3-709)

To aid recovery from aborted jobs, we recommend that *form feed* strings always contain a form feed, *page end* strings a full printer reset, and *end of text job* strings both a form feed and full printer reset.

General points, and Epson and IBM compatible printers (page 3-711)

The printer type is used to differentiate between printer definitions. If you try to overload a printer definition with one having the same printer type, the old data is retained. This avoids any delays that might occur if the user tries to load the same file twice.

It follows that if you make minor alterations to a definition and wish to load it in place of or beside the original, you must change the printer type.

Loading and setting the current territory (page 3-795)

The description in this section is wrong, and should read:

Each computer running RISC OS has a configured value for the current territory, set using *Configure Territory (see page 3-854), and stored in its CMOS RAM. On a reset or a power-on, RISC OS will try to load this territory as follows:

1 It will load any territory modules in ROM. (Typically there is only one, for the territory into which the computer has been sold.) If one of these is the configured territory, no further action is taken.

2 Otherwise, it will look on the *configured device* (ie the configured filesystem and drive) for the module &.!Territory.Territory, and load it.

3 If successful, it will then search for the directory …!Territory.Messages, and load any modules it contains. The directory should exist, even if it contains no modules.

At the end of this process:

- If the configured territory is in ROM, only those territory modules in ROM will be loaded
- If the configured territory is not in ROM, both those territory modules in ROM and another territory module (hopefully the configured one) will be loaded.

RISC OS then selects as the current territory either the configured territory, or – if it is not present – a default territory from ROM.

Sound_Speaker (page 4-24)
*Speaker (page 4-64)

These commands may not work on all machines, particularly those that use the headphone socket to mute the loudspeaker.

Squash_Compress (page 4-104)
Squash_Decompress (page 4-106)

The input and output pointers for these calls must be word-aligned.

_kernel_swi (page 4-283)
_kernel_swi_c (page 4-283)

If you use these functions to call a SWI that returns an error longer than 148 bytes, the register dump area is corrupted; even longer errors may corrupt other vital system data. You should ensure that no error will be returned – or workround this problem by instead using the internal function `_swix`, which is documented in the C library header files.

*Obey (page 4-358)

Recursive calls of *Obey are only possible to a limited depth (currently 20, although you should not rely upon this).

Draw files (page 4-463)

There are some errors in the documentation of Draw file formats, as follows:

- The font table object (page 4-465) may contain multiple <font number, font name> pairs, which follow immediately after each other; ie the padding to a word boundary only occurs at the end of the object.

  In RISC OS 3.5 and earlier, the Draw application expects the font table object to be the first object in the file; we suggest that any Draw files you generate obey this restriction. From RISC OS 3.6 onwards, Draw merely expects that the font table object precedes any text objects or transformed text objects that use it.
- The translation part of the transformation matrix must be zero for a transformed text object (page 4-474).
- The description of transformed sprite objects (page 4-475) should refer to ‘EIG factors’, not to ‘eigen factors’.

Font files (page 4-476)

There are some errors in the documentation of font file formats, as follows:

- The heading *IntMetrics / IntMet*n *files* on page 4-476 should read *IntMetrics / IntMetric*n *files.*
- The section entitled *Scaffold data* on page 4-484 should start:

Size	**Description**
1 or 2	character code of ‘base’ scaffold entry (0 ⇒ none)

- In the section entitled *Character data* on page 4-486, the lines:

  If *character flags* bit 3 is set:
  bit 4 set ⇒ composite character
  bit 5 set ⇒ with an accent as well

  would be clearer were they to read:

  If *character flags* bit 3 is set:
  bit 4 set ⇒ composite base character follows
  bit 5 set ⇒ composite accent character follows

  On the next page, the line:
  if *character flags* bits 3 or 4 are clear:
  should read:
  if *character flags* bit 3 is clear, or bit 3 is set and bits 4 and 5 are clear:

  and the final line of the section:
  Word-aligned at the end of the character data.
  should read:
  Word-aligned at the end of the chunk.

Printer server protocol interface

The printer server protocol interface was omitted from the *RISC OS 3 Programmer's Reference Manual*. It is currently as follows.

NetPrint status protocol

Status enquiry packet

To request the current state of a printer server the client sends an 8 byte status enquiry packet to port &9F:

Byte	Meaning
1 - 6	printer name, padded with spaces
7	reason code (1 ⇒ status request, 6 ⇒ name request)
8	reserved (must be zero)

Status request

If the reason code is 1 (status request) the printer server should check the printer name. The check should be case insensitive, but with accents significant, preferably using Territory_Collate (see page 3-842):

- If the name matches the name of a printer connected to the server (eg 'PScrpt'), the server should send its status.
- If the name matches the string 'PRINT' or 'SPOOL', the server should send the status of the user's default printer. (With Acorn's !Spooler software, this is the most recently used printer, or the first listed printer if none has yet been used).
- If the name matches neither of the above cases, the server should not reply.

The status reply, if any, must be sent to port &9E:

Byte	Meaning
1	status: 0 ⇒ Ready, 1 ⇒ Busy, 2 ⇒ Jammed, 6 ⇒ Offline; all other values reserved
2	station number for Busy status, or 0
3	net number for Busy status, or 0

If the server is Busy, the second and third byte of the status packet are the station and net number with which it is busy. If the server is Busy with no particular station, or if the status is not Busy, these bytes should both be set to zero.

Using the name 'PRINT' is deprecated because it makes it difficult for a printer server that supports multiple logical printers. Wherever possible you should use the printer's name.

Name request

If the status enquiry reason code is 6 (name request) then the client is asking the printer server for its name. The name sent by the client is 'PRINT' or 'SPOOL', but it is not necessary to check this. The server must reply to port &9E:

Byte	Meaning
1 - 6	printer name, padded with spaces

If the printer server supports multiple logical printers it may send multiple replies with different names. If the client discards duplicate replies then it should take account of the name in the packet as well as the station and net numbers.

Flag bytes

For all status packets the flag byte currently has no meaning. Clients should send a flag byte of zero, and servers should send back the flag byte that they received from the client.

NetPrint printing protocol

Finding the status before printing

Before starting to print, the client should ideally send a status enquiry to the server to ensure it is ready (see above).

Establishing the connection

The connection is then established using packets where the flag byte is relevant. It has this meaning:

Bits	Meaning
0	sequence bit
1, 2	modes
3 - 6	task id
7	reserved (must be zero)

The client first sends a zero or one byte packet to port &D1 on the server, with the flag byte's sequence bit clear, and its mode bits set to 2_01. The task id bits of this packet's flag byte – and its data – are used to negotiate how to send the print data. Their possible values are dependent on the version of NetFS in use, and are as follows:

- **If the flag byte's task id is 2_0000**, then the client will only send data in &50 byte blocks.

  If any byte is sent it should be zero, but is ignored.

- **If the flag byte's task id is 2_1000**, then the client code is both asking for the allocation of a task id by the server, and trying to establish if the server can accept large blocks of data (up to the size returned by SWI Econet_PacketSize) or only small ones (up to &50 bytes).

  If a non-zero byte is sent, the client is also seeking to negotiate a features mask with the server. The bits show the features the client supports:

Bit	Meaning when set
0 †	Use reply port &D0 (allows local loopback etc to work)
1	Print data is compressed (not yet implemented)
2	Use dynamic port for data packets
3 - 6	Reserved
7	More features in extension packet (not yet implemented)

  † This bit must always be set if any other bits are set.

- Other values of the flag byte's task id are reserved.

If the server is unwilling to accept the print it doesn't send a reply. If it is willing then it replies as follows:

- **If the client's task id was 2_0000**, the server sends back a single zero byte to port &D1, with the flag byte the same as that it received from the client.
- **If the client's task id was 2_1000**, the server uses the flag byte to respond to the request for large packets and task id…
  - If the server isn't willing to assign task ids – and hence accept more than one connection from a single client – it sends back the client's (illegal) task id of 2_1000 (see below); otherwise it sends back a task id chosen from the ranges 2_0001 to 2_0111, or 2_1001 to 2_1111.
  - If the server can accept large blocks of data it sets the mode bits to 2_10, else it sets them to 2_01.

  …and it uses the byte(s) it sends back to respond to any request for a features mask:
  - If the client did not request a features mask, or the server does not support any features, it sends back a single zero byte to port &D1.
  - If the client requested a features mask, and the server supports this, it ANDs its own mask with that sent by the client. If bit 2 is clear, the server sends the single mask byte to port &D0; if it is set, the server gets a dynamic port using Econet_AllocatePort, and sends two bytes to port &D0: the mask followed by the port.

The connection is now established. The client then examines the final flag byte sent by the server, changing a task id of 2_1000 to 2_0000. This version of the flag byte is the one that will be used when sending the data.

Sending the data

The client then sends the data in blocks, the size of which can vary from zero bytes up to the maximum established by the connect protocol. The data is sent to the dynamic port returned at connection time, if any; otherwise it is sent to port &D1. The flag byte for each block is the same as that negotiated when connecting (see above), save that the sequence bit is toggled for each block. This is to avoid duplicate data packets; the server discards and ignores any packets that have the same sequence bit as that previously received.

Acknowledging the data

Each time the server receives a new block and is ready to accept another, it must acknowledge the received block with a one byte packet. If the features mask negotiated in the connect protocol had bit 0 set, the reply is sent to port &D0, and the byte gives the status:

Value	Meaning
0	Ready (send next data packet)
1	Busy (don’t send next data packet yet)

Otherwise the reply is a zero byte sent to port &D1.

The packet’s flag byte must match that received from the client. Again, the sequence bit is used to avoid duplicates; if the flag byte of an acknowledgement received by the client does not match the packet it most recently sent, it is a duplicate of a reply to the previous packet, and so is discarded.

Closing the connection

When the client wants to close the connection, it sends a data packet with the mode bits set to 2_11. The data for this last packet must be terminated by an &03.

Port claiming

NetPrint claims ports &D0, &D1 and &9E with Econet_ClaimPort. A printer server should claim port &9F.

Deprecated calls

This section lists calls, often provided for backwards compatibility, that are now deprecated in favour of other calls. Much of this information is already in other parts of the PRM, but has been gathered together for reference.

VDU calls

Many of the VDU calls that are present in RISC OS have been superseded by either the OS_Plot call or other SWIs. Instead of using the VDU call, you should call the relevant SWI.

Examples

- You should use OS_Plot instead of VDU 25.
- You should use the standard printer driver interfaces to direct output to the printer, instead of calling VDU 2 and VDU 3.
- You should ColourTrans SWIs to set text and graphics colours instead of calling VDU 17 and VDU 18.
- You should use the font manager instead of calling VDU 23,25-26.
- You should use OS_SpriteOp SWIs instead of VDU 23,27

OS_Byte/OS_Word calls

Many of the OS_Byte and OS_Word calls are very archaic, and are only present in RISC OS for backwards compatibility with older 8 bit machines. Many of these calls have been superseded by RISC OS SWIs which you should use instead.

It is worth noting that many of the OS_Byte calls are either not necessary or there are SWI equivalents. In future versions of the operating system OS_Byte may be removed altogether, and the useful calls be coded as proper RISC OS SWIs. The same applies to OS_Word calls.

OS_Byte examples

- OS_Byte 7 and 8 are used to specify the serial port’s baud rates for receiving and sending data. These calls have been superseded by OS_SerialOp 5 and 6.
- OS_Byte 128 is used for reading the position/state of the mouse. It has been superseded by OS_Mouse.
- OS_Byte 71 selects the keyboard or alphabet. It has been replaced by the concept of territories. You should call the Territory manager for doing this sort of operation.

- All the OS_Bytes that refer to buffers (such as 15 to flush a buffer) have been replaced by the relevant software vectors.
- The OS_Byte calls that refer to the escape key (such as 125 to set Escape condition) are usually irrelevant, and should not be used on a multi-tasking operating system. An exception is OS_Byte 229, which may be useful to temporarily alter the Escape key status between successive Wimp polls.
- OS_Byte 143 should not be used for issuing service calls; OS_ServiceCall should be used instead.
- OS_Byte 160 reads a VDU variable; it has been superseded by OS_ReadVduVariables.

OS_Word examples

- OS_Word 9 should no longer be used to read the logical colour of a pixel. You should use OS_ReadPoint instead.
- OS_Word 11 should no longer be used to read the palette. OS_ReadPalette should be used instead.
- OS_ReadVduVariables should be used instead of OS_Word 13 to read current and previous graphics cursor positions.
- OS_Word 21,0 should no longer be used for setting the pointer shape etc. You should use OS_SpriteOp 36 (set pointer shape) instead.

FileSwitch

Service_StartUpFS has been removed.

As noted before, OS_Byte calls are deprecated. For example:

- OS_Byte 127 is deprecated, and you should use OS_Args 5 instead.
- You should no longer use OS_Byte 139 to set filing system options. *Opt 1 is no longer supported anyway. For the *Opt 4 usage you should instead use OS_FSControl 48. (This is in preference to OS_FSControl 10 which – although it is the direct equivalent – requires some state to be set up with the *Dir command before calling it.)

Many OS_GBPB calls are also deprecated:

- You should not use OS_GBPB 5 to read the name and boot option of a disc. You should instead use OS_FSControl 37 (canonicalise path) and/or OS_FSControl 47 (read boot option),
- You should no longer call OS_GBPB 6 or 7 to read a directory name and privilege byte. OS_FSControl 37 (canonicalise path) provides an alternative for reading directory names; privilege bytes are no longer supported.
- You should use OS_GBPB 9 in preference to OS_GBPB 8.

Finally, as hinted above, you should use OS_FSControl 48 in preference to OS_FSControl 10.

System extension/application SWIs

RISC OS implements many SWIs for application and system extension (ie modules) development. Although theses SWIs are present and usable in the OS, some of them are archaic and have alternatives that should be used.

Econet

With the event of AUN, most of the immediate operations are no longer supported. The only immediate operation supported under AUN is Econet_MachinePeek. If an application wishes to be AUN compatible then they should not attempt to implement the other immediate operations.

Time and date

You should no longer use SWIs such as OS_ConvertDateAndTime and OS_ConvertStandardDateAndTime. You should instead use the SWIs provided by the Territory manager.

Font Manager

When scanning a string for information (eg the width of the string or the caret position) you should call Font_ScanString instead of calls such as Font_StringWidth, Font_Caret, Font_StringBBox etc. However, Font_ScanString is a RISC OS 3 only SWI.

When setting font colours you should use ColourTrans_SetFontColours instead of Font_SetFontColours.

When calling Font_Paint with control sequences to set the colour, you should use control sequence 19 instead of 17 and 18. Again, control sequence 19 is only available with RISC OS 3.

You should not normally use the calls Font_SetFontMax (and the equivalent *Configure FontMax), Font_ReadFontMax, Font_SetScaleFactor, Font_ReadScaleFactor, and Font_SetThresholds. In doing so, you would be overriding the values set up by users and/or managed by the Wimp.

ColourTrans

Applications should not use GCOLs; they should instead deal with RGB palette entries and colour numbers.

If you must set a GCOL you should call ColourTrans_SetGCOL, or ColourTrans_ReturnColourNumber and OS_SetColour; you should not call ColourTrans_ReturnGCOL and then set the colour.

SWI Calls

OS_SerialOp 7 (SWI &57)

This reason code is for system use only; you must not use it in your own code.

OS_SerialOp 8 (SWI &57)

Read/write serial input buffer threshold value

On entry

R0 = 8 (reason code)
R1 = –1 to read or new value to write

On exit

R0 preserved
R1 = value before being overwritten

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 serial input routine attempts to halt input when the amount of free space left in the input buffer falls below a certain level. This call allows the value at which input is halted to be read or changed.

OS_SerialOp 0 can be used to examine or change the handshaking method.

The default value in RISC OS 3.5 is 17 characters, but this is subject to change and should not be relied upon.

Related SWIs

OS_Byte 203 (page 2-462)

Related vectors

SerialV