Contents

About this manual

Summary of contents

This manual gives you information about some of the technology to be used in a future version of the RISC OS operating system. This operating system has been designed to be used with some new hardware technology not currently in use.

This manual is designed to be used in conjunction with the RISC OS 3 *Programmer's Reference Manual*, and is produced as volume 5 in the set.

The manual only documents the differences between RISC OS 3.1 and a future version of RISC OS. 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.

We've laid out the information in these parts as consistently as possible, to help you find what you need. Each chapter covers a specific topic, and in general includes:

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

Indexes

The Indexes at the end of this volume include references to this volume and the four previous volumes. This enables you to find all references to a topic at a single glance. Index entries for this volume are given in bold. These are:

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

- an alphabetic index of SWI calls
- an index by subject.

Conventions used

Certain conventions are used in this manual:

Hexadecimal numbers

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

&FFFF (65535)

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

Typefaces

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

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

Command syntax

Special symbols are used when defining the syntax for commands:

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

Programs

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

- BBC BASIC examples omit any line numbering
- BBC BASIC Assembler programs do not show the structure needed to perform the assembly

- ARM Assembler programs assume that header files have been included that define the SWI names as manifests for the SWI numbers.
- 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 Desktop* C and *Acorn Desktop Assembler* products, which you can use to write programs in (respectively) C and ARM assembler. Both products run in a desktop environment with full supporting tools. The manuals for both products are available separately if required: they are entitled *Acorn* ANSI C *Release* 4 and *Acorn Assembler Release* 2.

The hardware is described in the *Technical Reference Manual*.

Reader comments

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

1 Introduction to RISC OS 3.X

Introduction

RISC OS 3.X is an operating system written by Acorn for its new architecture. The operating system has only been changed where it has been necessary to support the changing hardware. We have tried to make it as compatible as possible with previous versions of the operating system.

This manual just documents the changes made in the programmers interface for RISC OS 3.X. You still need your copy of the RISC OS 3 *Programmer's Reference Manual*.

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 supplied with the new architecture is called RISC OS 3.X in this manual.

Hardware overview

The main electronic components of the computer are:

- An ARM (Advanced RISC Machines) ARM610 or ARM700 processor, which provides the main processing in the computer.
- A VIDC20 (Video Controller) chip, which provides the video and sound outputs of the computer.
- 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.

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

This diagram gives a schematic of an architecture which may be viewed as typical of this series of computers.

ARM 6xx and ARM 7xx

The ARM is a RISC (Reduced Instruction Set Computer) processor.

With this generation of computers Acorn have two different versions of the ARM processor available.

The ARM 6xx delivers about x times the power of an ARM 2 (XX million instructions per second, or MIPS, compared to some 4 - 5 MIPS for the ARM 2).

The ARM 7xx delivers about x times the power of an ARM 2 (XX million instructions per second, or MIPS, compared to some 4 - 5 MIPS for the ARM 2). The ARM 7xx 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 ARM 7xx supports the same instruction set as the ARM 6xx.

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. You should refer to the RISC OS 3 *Programmer's Reference Manual* for information about VIDC1. This section only refers to the differences between VIDC1 and VIDC20.

The 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

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 paletter 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 is capable of generating a display at any pixel rate up to xxxMhz. 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 palette is compatible with to the VIDC1 sound system with an independant 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.

See the xxSound Chapter pagexx of the RISC OS 3 PRM

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, I^2C, 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 ARM6xx/ARM7xx 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.

RISC OS 3.X overview

RISC OS 3.X supports the new hardware architecture. This includes support for:

- The ARM6xx and ARM7xx CPUs and associated floating point hardware.
- The VIDC20 video chip and its 16 bit and 24 bit colour modes.
- A new mode picker interface to choose high resolution mode.
- A new colour picker for choosing colours.
- A new sprite format.
- The PC keyboard interface to allow a standard PC keyboard to be used.
- New faster and more efficient serial and parallel device driver software.
- Connecting a PC-type serial mouse to the serial port.
- DMA access to various ports and expansion cards.
- A new network expansion card interface.

The following sections give a quick overview of the main changes in RISC OS 3.X.

Memory management

Memory management now incorporates the following:

- Up to 256MB DRAM and 2MB VRAM memory allowed.
- Direct memory access (DMA) control.
- Any second processor card can claim a chunk of memory.
- The physical RAM allocation does not have to be contiguous.
- The ARM6xx page table allocation.
- Expansion of the logical memory map 32 bit address space.

More information about memory management can be found starting on page 9.

DMA

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

The DMA module:

- Performs the arbitration and switching between devices (with help from the device drivers).
- Provides a general purpose software interface to the DMA channel hardware.
- Isolates hardware from software so that changes to the hardware only affect the DMA manager and not DMA clients.
- Handles memory mapping and memory management, so that any DMA clients are not concerned with logical to physical addres translation 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.

Parallel and serial device drivers

The serial and parallel device drivers have been changed to:

- Speed up the performance of these ports.
- Take advantage of new hardware.

The parallel device can be opened either for input or output, but not for both at the same time.

Video

The video system in the new architecture has been substantially changed so that the new VIDC20 video controller chip can be used to its full capabilities.

Using VIDC20 gives a much improved video capability over the previous generation of RISC OS computers that used VIDC1 chips.

VIDC20 allows 16 bpp (*bits per pixel*) and 32 bpp screen displays as well as all previously available colour depths. Additionally there is also an 8 bpp grey-level mode available.

These colour depths can be translated into numbers of colours:

8 bpp	256 colours
16 bpp	32 thousand colours (actually 32,768)
32 bpp	16 million colours (actually 16,777,216)

Colour picker

The colour picker module is a utility that allows users to pick a colour from the full range available. This utility can be used by all future applications that need to choose colours. It is currently used by !Draw, !Paint and several third party applications.

Keyboard and mouse

One of the main change in the new architecture is the removal of the Acorn keyboard interface from the kernel and its replacement with a standard PC AT-style keyboard driver provided as a separate module.

The standard quadrature mouse driver is now a module.

There is also a serial mouse driver module that allows the use of a standard PC-type mouse connected to the serial port.

Expansion card support

The new architecture expansion card interface has been enhanced in several ways. It now supports (in addition to the existing facilities):

- 32 bit wide data paths
- A mew 16MB address space for each card
- A dedicated Network card interface
- Direct Memory Access (DMA).

Only cards designed for the new architecture will be able to take advantage of these features.

Acorn universal networking

Acorn universal networking software (AUN) is provided as part of the operating system.

2 Memory management

Introduction

This chapter describes the changes in memory management in RISC OS 3.X. These changes have been caused by the changes in the underlying hardware used in the new architecture.

Memory management now incorporates the folowing:

- Up to 256MB DRAM and 2MB VRAM of memory allowed.
- Improved direct memory access (DMA) control.
- Any second processor card can claim a chunk of memory.
- The physical RAM allocation does not have to be contiguous.
- The ARM6xx page table allocation.
- Expansion of the logical memory map due to the 32 bit address space.
-

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 model then used a free pool and multiple application spaces, rather than just the one application space which had all spare memory in it.

In the new architecture the free pool memory is managed by the kernel. The Wimp manages the multiple application spaces; it is responsible for constructing and managing tasks in the desktop.

A new SWI OS_DynamicArea is used for the control, creation and deletion of dynamic areas. The existing SWI OS_ChangeDynamicArea, which allows the resizing of dynamic areas, is used unchanged.

To cope with the new maximum memory size on the new architecture a new dynamic area is used, known as the "free pool" (area number 6).

How the free pool operates

- 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.
- If the free pool itself 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.

Window Manager module

The Window Manager understands the free pool. Its operation is simplified, as it no longer needs to maintain its own free pool. The SWIs Wimp_TransferBlock and Wimp_ClaimFreeMemory have been modified to reflect this.

Dynamic areas

In RISC OS 2 and 3 (version 3.10) the main kernel interface for memory management was OS_ChangeDynamicArea, which allowed the resizing of dynamic areas. The change dynamic area SWI called other modules depending on which dynamic area was being resized. The result was that there was no flexibility with how dynamic areas were used.

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.

In the new architecture the SWI OS_DynamicArea is used for the control, creation and deletion of dynamic areas.

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 (that IOMD presents to the ARM processor core).

Logical address

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

Physical page number

An arbitrary number assigned to each page of RAM physically present in the computer.

Page blocks

Several interfaces use Page blocks. These are tables of 3-word records

Word	Use
0	Physical page number
1	Logical address
2	Physical address

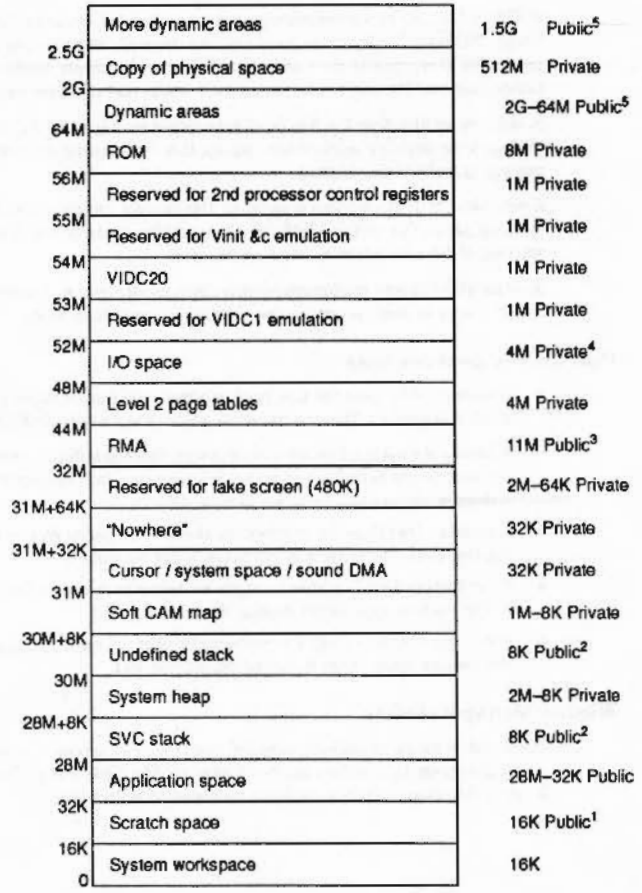
A block of memory 12*N bytes big where N is the number of records in the block. They are used to pass round lists of addresses or pages.

User Interface

The user interface for the Task manger window has changed slightly to cope with the increased maximum memory size and decrease in page size. For more information see the chapter entitled *User interfaces* on page 293.

Technical Details

Logical memory map



Notes about the logical memory map:

1 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 uses 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 Public[2] areas can be assumed to end on a 1MB boundary. An exception will occur if they are accessed beyond this point. The exact location of these stacks should not be assumed.

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

4 Private[4] areas are Private and used for I/O except where device drivers export hardware addresses.

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

Free pool

To cope with the new maximum memory size on the new architecture a new dynamic area is used, known as the *free pool* (area number 6). The SWI OS_ChangeDynamicArea has been changed to reflect this, this has already been described on page XX.

WIMP changes

The Window Manager has been modified to use the free pool. In particular, the programmer's interface to the following SWIs have changed:

Wimp_TransferBlock

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.

It now splits the transfer into chunks of no more than half of the maximum application memory size. For each chunk, it puts the appropriate pages of the source block in at the start of application space, and the appropriate pages of the destination block above these. It then performs the transfer. After all chunks have been transferred, it restores the current task's pages in application space.

Wimp_ClaimFreeMemory

Wimp_ClaimFreeMemory has also been modified, because the Wimp no longer maintains control of the free pool. It creates a new dynamic area of the size requested by the caller and passes that address back. When the memory is freed the area is removed.

New SWIs

The following new SWIs have been created, they are defined in full at the end of this chapter starting on page 22.

- The SWI OS_DynamicArea is used for the control, creation and deletion of dynamic areas. R0 provides a reason code which determines which operation is performed.
- As all operations on dynamic areas work in physical page numbers it is not possible to map anything other than RAM pages (DRAM and VRAM) into a dynamic area. In particular the extension to the existing expansion card bus space, known as the EASI *space*, cannot be mapped in.
- OS_Memory. This SWI performs miscellaneous operations for memory management.

  Here are the defined reason codes:

R0	meaning	page
0	general page block operations	page 38
1 to 5	reserved	page 40
6	read physical memory arrangement table size	page 41
7	read physical memory arrangement table	page 42
8	read amounts of various sorts of memory	page 43
9	read controller presence	page 44

- OS_ClaimProcessorVector allows a module to attach itself to one of the processor's vectors.

Changes to Existing SWIs

OS_SetMemMapEntries (SWI &53)

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

OS_ReadDynamicArea (SWI &5C)

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 in RISC OS 3.X.

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.

OS_Heap (SWI &1D)

Reason code 0 (Initialise heap):
In the RISC OS 3 PRM it states that 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.

In RISC OS 3.x platforms the only restrictions are that the base of the heap must be word-aligned, and the size must be a multiple of 4 bytes.

*Cache change

*Cache now switches both caching and write buffering on and off. The corresponding SWIs are not implemented.

Dynamic area handler routine

This section describes the reason codes passed to the dynamic area handler routine, with their entry and exit conditions. This routine is called when the size of an area is being changed.

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.

Reason codes

On entry, R0 contains a reason code, which describes what is happening. The reason codes are as follows:

PreGrow (0)

Issued when a call results in an area growing.

On entry

R0 = 0 (reason code)
R1 = pointer to a page block. The physical page number entries are set to -1
R2 = number of entries in Page block (number of pages area is growing by)
R3 = number of bytes area is growing by (R2 * the page size R5)
R4 = current size of area (bytes)
R5 = page size
R12 = pointer to workspace

On exit

If the growth is OK, then

R0 is preserved

If particular physical page numbers are required then all the physical page number entries must be filled in with the required pages. The other entries must be left alone.

```
    V = 0
else
    R0 -> error to return, or zero to return generic error
    V = 1
endif
```

All other registers preserved

Use

This reason code is issued when a call to OS_ChangeDynamicArea results in an area growing.

It permits the dynamic area handler to request that specific pages be used for growing the area. If this is the case then all pages must be specified. The correspondence between the Page block and memory is that 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.

It is called before any pages are actually moved.

If an error is returned, then the area will not change size.

PostGrow (1)

Issued when a call results in an area growing.

On entry

R0 = 1 (reason code)
R1 = pointer to a page block. Only the physical page number entries are defined.
R2 = number of entries in Page block (number of pages area grew by)
R3 = number of bytes area grew by (R2 * the page size R5)
R4 = new size of area (bytes)
R5 = page size
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 when a call results in an area shrinking.

On entry

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

On exit

If shrink (even by reduced amount) is OK, then

R0 preserved
R3 = number of bytes area can shrink by. This must be less than or equal to R3 on entry.
    V = 0
else
    R0 -> error block, or zero to return generic error
    R3 = 0
    V = 1
endif

All other registers preserved

Use

This reason code is issued when a call to OS_ChangeDynamicArea results in an area shrinking. It is called before any pages are moved. It allows the handler to limit the amount of memory moved out of the area, or to object to the size change altogether. The shrink amount allowed as returned by this reason code is permitted to be a non-page multiple. The ChangeDynamicArea code will ensure the shrink permitted is always rounded down to a page multiple before it is actioned.

PostShrink (3)

Issued when a call results in an area shrinking.

On entry

R0 = 3 (reason code)
R3 = number of bytes area shrunk by
R4 = new size of area (bytes)
R5 = page size
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

This section has been provided to give an overview of what happens when a dynamic area's size is changed. This is presented as pseudo-code for clarity.

```
Check IRQSemaphore - reject CDA if set

Growing free pool:
(no check for page availablility - do as much as possible)
Application space being shrunk - confirm it's OK:
    IF CAO in application space THEN
        UpCall_MovingMemory (asks application if it consents to memory move)
        IF UpCall *not* claimed THEN reject CDA
    ELSE
        Service_Memory (asks modules for objectors to the memory move)
        If Service *is* claimed THEN reject CDA
    ENDIF
Move pages from application space end to free pool

Growing other dynamic area:
    Check for page availability -if not enough available bounce CDA with error
    Table of pages prepared:
        Allocates memory
        Fills in -1's for 'any page'
    PreGrow is called:
        replaces -1's if it wants
        objects about resize amount perhaps
    Check for unavailable pages:
        IF there is a non -1 which can't be grabbed THEN reject CDA
    Check for application space resizing:
        IF free pool < amount needed THEN
            IF CAO in application space THEN
                UpCall_MovingMemory (asks application if it consents to
                                     memory move)
                IF UpCall *not* claimed THEN reject CDA
            ELSE
                Service_Memory (asks modules for objectors to the memory move)
                IF Service *is* claimed THEN reject CDA
            ENDIF
        ENDIF
    Page replacements determined:
        Work out swap sequences on all non -1's
        Replace all -1's with actual pages
        Pages get grabbed first from the free pool, then underflowing int the
                                                    application space
    Issue Service_PagesUnsafe (only if PreGrow specified pages)
    Pages get moved around:
        Do the page moving/swapping (don't swap if pages requested are in
                                     free pool)
    Issue Service_PagesSafe (only if PreGrow specified pages)
    PostGrow is called:
        Sorts out structures for the new size
```



```
Shrinking free pool:
Check if application space OK to grow:
    IF application space < maximum THEN
        IF CAO in application space THEN
            UpCall_MovingMemory (asks application to consent to memory move)
            IF UpCall *not* claimed THEN reject CDA
        ELSE
            Service_Memory (asks modules for objectors to the memory move)
            IF Service *is* claimed THEN reject CDA
        ENDIF
    ENDIF
Move pages from free pool to application space

Shrinking other dynamic area:
    PreShrink is called:
        objects about resize amount perhaps, or gives larger allowed size
        Sorts out structures for the new smaller size as the shrink will
                                                    definitely go ahead.
    Pages get moved around:
        Move pages from dynamic area to free pool
    PostShrink is called:
        Keep subsystem informed.
```

The system stack is used for the page structure passed to the PreGrow routine. 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 and Service_PagesSafe. 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 &xx)

Pages specified have been swapped around

On entry

R1 = Service_PagesUnsafe
R2 = frame filled in by the PreGrow routine with the two address fields filled in too.
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 around. Direct memory access activities involving the specified pages should be suspended until Service_PagesSafe has been received indicating the pages are safe.

Service_PagesSafe (Service Call &8F)

Pages specified have been swapped for different pages.

On entry

R1 = Service_PagesSafe
R2 = number of entries in each Page block
R3 = Page block before move
R4 = Page block after move

On exit

All registers preserved

Use

This service call informs recipients that the pages specfied 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 replaced, and the 'after' block the page numbers and physical addresses of the pages which replaced them.

Service_DynamicAreaCreate (Service Call &90)

Dynamic area has just been created.

On entry

R1 = Service_DynamicAreaCreate (&90)
R2 = area number of 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.

This service call must not be claimed

Service_DynamicAreaRemove (Service Call &91)

Issued just before the removal of a dynamic area.

On entry

R1 = Service_DynamicAreaRemove (&91)
R2 = area number of area about to be removed

On exit:

All registers preserved

Use

This service must not be claimed

This service is issued just before the removal of a dynamic area. It is issued during a call to OS_DynamicArea(1), after the area has been successfully reduced to zero size, but before it has been removed completely.

This service call is designed to keep 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.

Service_DynamicAreaRenumber (Service Call &92)

Issued during a call to OS_DynamicArea(2).

On entry

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

On exit

All registers preserved

Use

This service must not be claimed

This service is issued during a call to OS_DynamicArea(2), (when an area is being renumbered).

This service call is designed to keep 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.

SWI calls

OS_DynamicArea (SWI &66)

This SWI performs operations on dynamic areas. R0 provides a reason code which determines which operation is performed.

On entry

R0 = 0 (reason code)
other registers as determined by reason code

On exit

R0 preserved
other registers may return values, as determined by the reason code passed.

Interrupts

Processor Mode

Re-entrancy

Use

This call is a single call with many operations within it. The operation required, or reason code is passed in R0. It can have the following meanings.

R0	Meaning	Page
0	Creates a new dynamic area	page 28
1	Removes a previously created dynamic area	
2	Returns information on a dynamic area	
3	Enumerates a dynamic area	
4	System use only	
5	System use only	

OS_DynamicArea 0 (SWI &66)

This call creates a new dynamic area.

On entry

R0 = 0 (reason code)
R1 = new area number (-1 ⇒ RISC OS should allocate number. Must not be 128 to 255)
R2 = initial size of area (in bytes)
R3 = base logical address of area (-1 ⇒ RISC OS should allocate base address)
R4 = area flags

bits 0 to 3 = access privileges to be given to each page in the area (same format as for OS_Read/SetMemMapEntries)
bit 4 = 0 ⇒ area is bufferable
= 1 ⇒ area is not bufferable
bit 5 = 0 ⇒ area is cacheable
= 1 ⇒ area is not cacheable
bit 6 = 0 ⇒ area is singly mapped
= 1 ⇒ area is doubly mapped
bit 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).
bits 8 to 31 must be zero

R5 = maximum size of area, or -1 if the area should be capable of growing to the total RAM size of the machine
R6 is a pointer to the area handler routine, or 0 for no routine
R7 = workspace pointer to be passed in R12 on entry to area handler (should be 0 if R6 = 0) (-1 ≥ OS will pass base address of area as workspace pointer).
R8 is a pointer to the area description string (null terminated), eg. 'Font cache'. This string will be displayed in the Task Manager window.

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

Interrupts

Processor Mode

Re-entrancy

Use

OS_ReadDynamicArea on these areas always returns with R2 being the maximum area size if R1 is -1 on entry. When R1 is -1 on entry, RISC OS allocates an area number itself, and this is greater than or equal to 256. This means that OS_ReadDynamicArea on these areas will always return with R2 being the maximum size.

On entry, R3 holds the base address of the area. This must be aligned on a memory page boundary (to read the page size use OS_ReadMemMapInfo).

Singly and doubly mapped areas

For singly mapped areas the base logical address is the lowest logical address used by that area. The area grows by adding pages at the high address end.

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.

If R3 on entry is -1, then RISC OS allocates a free area of logical address space which is big enough for the maximum size of the area.

On entry, R6 points to the area handler routine which gets called with various reason codes when an area is grown or shrunk. If zero is passed in, 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 Dynamic area handler routine section in this chapter

On entry, R7 contains a value which is passed in R12 on entry to the area handler routine. This normally points at workspace needed by the handler. If -1 is passed in, then RISC OS uses the base address of the area as the workspace pointer. This is particularly useful if the entry value in R3 is also -1, ie the caller does not know what base address is allocated by RISC OS at the time the area is created.

The area is created initially with size zero (no pages assigned to it), and is then grown to the size specified in size R2, which involves the area handler being called in the same way as if OS_ChangeDynamicArea was called to grow the area.

The area is created with a maximum size equal to either the amount given in R5 on entry, or the total RAM size of the machine, if this is smaller. If R5 is –1 on entry, then the maximum size will be set to the total RAM size of the machine.

If R3 on entry is –1, then RISC OS allocates a free area of logical address space which is big enough for the maximum size of the area.

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

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
- the ability to create doubly-mapped areas

Applications should in general 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.

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.

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

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

Interrupts

Processor Mode

Re-entrancy

Use

Removes a previously created dynamic area.

Before the area is removed, RISC OS attempts to shrink it to zero size. This is done using ChangeDynamicArea. If the ChangeDynamicArea returns an error then the area will be grown back to its original size using ChangeDynamicArea and the remove dynamic area call will return with an error. If the ChangeDynamicArea to reduce the area to 0 size worked then the area will be removed.

Once the area has been removed Service_DynamicAreaRemove is issued to inform the rest of the system about this change.

OS_DynamicArea 2 (SWI &66)

Returns information on a dynamic area.

On entry

R0 = 2 (reason code)
R1 = area number

On exit

R2 = current size of area (in bytes)
R3 = base logical address of area
R4 = area flags
R5 = maximum size of area
R6 a pointer to an area handler routine
R7 = workspace pointer for area handler
R8 a pointer to an area description string (null terminated)

Interrupts

Processor Mode

Re-entrancy

Use

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 = area number, or -1

On exit

R1 = next area number, or -1

Interrupts

Processor Mode

Re-entrancy

Use

Enumerates dynamic areas.

This allows an application to find out what dynamic areas are defined. -1 is used to start the enumeration and -1 indicates that the enumeration has finished.

OS_DynamicArea 4 (SWI &66)

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

OS_DynamicArea 5 (SWI &66)

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

OS_Memory (SWI &68)

Performs miscellaneous operations for memory management.

On entry

R0 = reason code and flags
Bits 0-7 are the reason code
Bits 8-31 are the flags which may be specific to the reason code

The other registers are specific to the reason code.

On exit

The returned values are specific to the reason codes.

Use

This SWI performs miscellaneous operations for memory management.

Here are the defined reason codes:

R0	meaning	page
0	general page block operations	page 38
1 to 5	reserved	page 40
6	read physical memory arrangement table size	page 41
7	read physical memory arrangement table	page 42
8	read amounts of various sorts of memory	page 43
9	read controller presence	page 44

Reasons 6 to 8: Physical Memory

These are provided to enable a program to find out what physical memory there is and its arrangement. The first two calls provide complete information on the available memory. The information is provided in the form of a table, with each page of physical memory space having 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, ie it 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.

The third call gives a summary of available memory.

OS_Memory 0 (SWI &68)

General Page block Operations

On entry

R0 = flags

bit	meaning
0 - 7	reason code (0 - 5)
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
12	Logical address will be filled in when set
13	Physical address will be filled in when set
14 - 15	Cacheability control: 0 ⟹ no change 1 ⟹ no change 2 ⟹ disable caching on these pages 3 ⟹ enable caching on these pages
16 - 31	reserved - set to 0

R1 ⟹ Page block
R2 = number of entries in page block

On exit

R1 = Page block updated as necessary

Use

This reason code is used to convert between representations of memory addresses. The different memory spaces are logical memory, physical memory and physical pages.

The Page block is scanned and the specified operations applied to it. It is possible to do address conversions and control the cacheability on a per-page basis. If any page is found to be unconvertable or non-existent then an error will be returned and the cachability unaffected. Cachability is accumulated for each page.

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

If any page is made physically uncacheable, then the cache will be flushed befoore the SWI exits. it is not necessary to do any conversion when changing cacheability (ie. bits 11 to 13 may be clear).

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.

This will only handle RAM addresses. The address fields may be non-page aligned.

OS_Memory 1 - 5 (SWI &68)

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

OS_Memory 6 (SWI &68)

Read Physical Memory Arrangement Table Size

On entry

R0 = 6 (bits 8 - 31 clear)

On exit

R1 = table size (bytes)
R2 = page size (bytes)

Use

This returns information about the memory arrangement table.

OS_Memory 7 (SWI &68)

Read Physical Memory Arrangement Table

On entry

R0 = 7 (bits 8 - 31 clear)
R1 = pointer to table to be filled in

On exit

R1 = pointer to updated table

Use

This returns the physical memory arrangement table in the block of memory pointed at by R1.

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

OS_Memory 8 (SWI &68)

Read Amounts Of Various Sorts Of Memory

On entry

R0 =	bits	meaning
	0 - 7	must be 8 – the reason code
	8 - 11	the type of memory: 1 ⇒ DRAM 2 ⇒ VRAM 3 ⇒ ROM 4 ⇒ I/O
	12 - 31	reserved - set to 0

On exit

R1 = number of pages of that sort of memory
R2 = page size (in bytes)

Use

Used to read in the types of memory available in the computer.

OS_Memory 9 (SWI &68)

Read Controller Presence

On entry

R0 = 9 (bits 8 - 31 clear)
R1 = controller ID

bit	meaning
0 - 7	controller sequence number
8 - 31	controller type:
	0 ⟹ EASI card access speed control
	1 ⟹ EASI space
	2 ⟹ VIDC1
	3 ⟹ VIDC20

On exit

R1 = controller base address or 0 if not present.

Use

These give information about the I/O space. Controllers are identified by type and sequence number so that a machine could be constructed with, say, more than one IDE controller in it.

This returns the location of a controller on the given machine. For example the EASI space 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 get at this information and to control your expansion card's EASI space access speed.

OS_ClaimProcessorVector (SWI &69)

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 & 3)
	6 - IRQ
	7+ - reserved for future use
8	0=release, 1=claim
9-31	reserved, must be 0

R1=replacement value
r2=value which should currently be on vector (only needed for release)

On exit

R1=value which has been replaced (only returned on claim)

Use

This SWI provides a means whereby a module can 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.

3 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 and some bytes are reserved for each expansion card. 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.

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 1-363) allows you to read the CMOS memory directly, while OS_Byte 162 (page 1-365) can write to it.

RISC OS 3.X allocation

The full usage of CMOS RAM in RISC OS 3.X is given below. Locations marked '†' were not used, or used for a different purpose in RISC OS 3.1.

Location	Function	
0	Econet station number (not directly configurable)	
1	Econet file server station id (0 ⇒ name configured)	
2	Econet file server net number (or first char of name – rest in bytes 153 - 157)	
3	Econet printer server station id (0 ⇒ name configured)	
4	Econet printer server net number (or first char of name – rest in bytes 158 - 172)	
5	Default filing system number	
6 - 7	*Unplug for ROM modules: 16 bits for up to 16 modules	
8 - 9	Reserved for Acorn use	
10 †	Screen info:	
	Bits 0 - 4	reserved for Acorn use
	Bits 5 - 7	TV vertical adjust (three-bit number)
11 †	Misc configuration:	
	Bits 0 - 2	ADFS drive
	Bits 3 - 5	001 ⇒ ShCaps, 010 ⇒ NoCaps, 100 ⇒ Caps
	Bit 6	loads directory on switch-on

0 = directory, 1 = No Directory

Bit 7 Not used

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 destination (configure print) print is sent to PrinterTypeSn

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 ⇒ S.ArthurLib
- Bits 2 - 3 FS list display mode: 0 ⇒ large icons, 1 ⇒ small icons, 2 ⇒ full info, 3 reserved
- Bits 4 - 7 reserved for Acorn use

18 - 19 *Unplug for ROM modules: 16 bits for up to 16 modules

20 - 21 *Unplug for extension ROM modules: 16 bits for up to 16 modules

22 Wimp double-click move limit

23 Wimp auto-menu delay

24 Territory

25 Printer buffer size

26 IDE disc auto-spindown delay

27 Wimp menu drag delay

28 FileSwitch options:

- Bit 0 truncate names: 0 ⇒ give error, 1 ⇒ truncate no error
- Bit 1 DragASprite: 0 ⇒ don't use, 1 ⇒ use
- Bit 2 interactive file copy: 0 ⇒ use, 1 ⇒ don't use
- Bit 3 Wimp's use of dither patterns on desktop: 0 ⇒ don't use, 1 ⇒ use
- Bit 4 reserved for Acorn use
- 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 options:

- 0 = standard quadrature mouse
- 1 = MicroSoft compatible serial mouse
- 2 = Mouse Systems Corporation compatible serial mouse
- 3 - 255 Reserved

30 - 45 Reserved for the user

46 - 79 Reserved for applications

80 - 111 Reserved for RISC iX

112 - 127 Reserved for expansion card use

128 - 129 Current year

130 - 131 Reserved for Acorn use

132 DumpFormat and Tube expansion card:

- Bits 0,1 control character print control: 0 ⇒ print in GSTrans format, 1 ⇒ print as a dot, 2 ⇒ print decimal inside angle brackets, 3 ⇒ print hex inside angle brackets
- Bit 2 treat top-bit-set characters as valid if set
- Bit 3 AND character with &7F in *Dump
- Bit 4 treat TAB as print 8 spaces
- Bit 5 Tube expansion card enable
- Bits 6,7 Tube expansion card slot (0 - 3)

133 Sync, monitor type, some mode information:

- Bits 0, 7 0 ⇒ vertical sync, 1 ⇒ composite sync, 3 ⇒ auto sync)
- Bit 1 reserved for Acorn use
- Bits 2 - 6 monitor type:
  - 0 = type 0 Normal
  - 1 = type 1 MultiFrequency
  - 2 = type 2 Hi-res Mono
  - 3 = type 3 VGA
  - 4 = type 4 Super VGA
  - 5 = type 5 LCD
  - 6 - 30 undefined
  - 31 = Auto

134 FontSize in units of 4K

135 ADFS byte 1

- Bits 0 - 2 number of floppy disc drives
- Bits 3 - 5 unused (was ST506)

†

Bits 6,7 number of IDE discs

136 ADFS byte 2
- Bits 0,1 step rate - floppy disc 0
- Bits 2,3 step rate - floppy disc 1
- Bits 4,5 step rate - floppy disc 2
- Bits 6,7 step rate - floppy disc 3

137 ADFS byte 3 - ADFS buffers
- 0 = no buffers
- 1 = number depends om memory size
- >1 = number of 1K buffers

138 Allocated to CDROMFS

139 TimeZone in 15min offsets from UTC, stored as signed 2's complement number (RISC OS 3 version 3.10 onwards)

140 Desktop features:
- Bit 0 3D: 0 ⇒ 2D look, 1 ⇒ 3D look
- Bits 1 - 4 Desktop font setting
  - 0 = Use WimpSFont and WimpSFontSize
  - 1 = Use System font
  - 2 - 15 = Use ROM font in ResourceFS
- Bits 5 - 6 reserved for Acorn use
- Bit 7 window background tiling enabled
  - 0 = tiled with tile_1
  - 1 = not tiles, grey 1

141 † No longer used – reserved for Acorn use

142 No longer used – reserved for Acorn use

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 voice 1-16 0 default voice
- Bits 4 - 6 loudness (0 - 7 ⇒ &01, &13, &25, &37, &49, &5B, &6D, &7F)
- Bit 7 not used – reserved

149 - 152 Allocated to BASIC Editor

153 - 157 Printer server name

158 - 172 File server name

173 - 176 *Unplug for ROM modules: 32 bits for up to 32 modules

177 - 180 *Unplug for expansion card modules: 4 x 8 bits for up to 8 modules per card

181 - 184 Wild card for BASIC editor

185 Configured language

186 Configured country

187 VFS

188 † Miscellaneous:
- Bits 0 - 1 ROMFS Opt 4 state
- Bit 2 cache icon enable state: 0 = no cache, 1 = cache 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 ISR
- Bit 3 User RPC
- Bit 4 OS RPC
- Bit 5 Halt
- Bit 6 GetRegs
- Bit 7 Undefined

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 (RISC OS 3 version 3.10 onwards)
- Bit 3 power saving: 0 ⇒ disabled, 1 ⇒ enabled
- Bit 4 mode and wimp mode: 0 ⇒ use byte 196, 1 ⇒ auto
- Bit 5 cache enable for ARM3: 0 enabled, 1 disabled
- Bit 6 broadcast loader protocols enable (0)
- Bit 7 broadcast loader colour hourglass enable (1)

196 Mode and Wimp mode

197 Wimp flags
- Bits 0 - 3 Instant dragging (1=allow, 0=disallow)
  - Bit 0 position
  - Bit 1 size
  - Bit 2 horizontal scroll
  - Bit3 vertical scroll

Bit 4 Action on error
0 = beep, 1 = don't beep
Bits 5-6 Allow windows off screen:
Bit 5 to bottom and right
Bit 6 to top and left
Bit 7 Auto open sub-menus: 0 = no auto, 1 = auto

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
0 = depends on memory
for >1MB 4x1K buffers for each MB fitted.

200 - 205 FontMax, FontMax1 - FontMax5

206 - 207 No longer used – 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 † No longer used – reserved

217-219 ROM unplug bytes 7 to 9

220 † Alarm and time byte
Bits 0-2 Format state:
000 ⇒ Illegal, !Alarm checks for first run
001 ⇒ Analogue with seconds
010 ⇒ Analogue without seconds
011 ⇒ HH:MM
100 ⇒ Format = "%24:%mi:%se"
101 ⇒ Format = "%z12:%mi:%se %am %zd %zmn %yr"
110 ⇒ Reserved
111 ⇒ Reserved
Bit 3 Deletion flag
0 ⇒ No confirmation

1 ⇒ Ask for confirmation
Bit 4 Auto save flag
0 ⇒ No auto save(D)
1 ⇒ Auto save
Bit 5 5 day week flag
0 ⇒ Disabled
1 ⇒ Enabled
Bit 6 Alarm noise
0= Not silent(D)
1 ⇒ Silent
Bit 7 Daylight Saving Time flag
0 ⇒ Normal time (non-DST)(D)
1 ⇒ Daylight Saving Time (DST)

221 Wimp Drag Time

222 Wimp Drag - Move Limit

223 Wimp Double - Click Time

224 - 230 † Reserved for RISC iX

231-232 † Reserved for Acorn use

233-237 † FSLock

238 † FSLock checksum

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 1-365 for more details.

4 DMA

Introduction and Overview

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 hardware from software so that changes to the hardware only affect the DMA manager and not DMA clients.
- 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 requests are processed first-come-first-served. It does not impose any priority on logical devices.

When a transfer is requested, the DMA manager 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, extended 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
4 Sound out
5 Sound in

The four general purpose physical channels must be shared by several devices via logical channels. The mapping between logical and physical channels is fixed. They have been assigned logical numbers for future use. Only podules 0 and 1 have DMA connected.

The following logical channels are supported.

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

Mapping between logical and physical channels

The four general purpose physical DMA channels can be connected to devices on either side of the expansion card buffer. The expansion card buffer is not be output enabled during DMA operations to internal peripherals, but is be 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 must interface in the following ways:

1 The DMA manager maps logical addresses to physical addresses so that the IOMD DMA registers can be programmed. This is done by creating a page table containing the logical addresses of all pages used in the transfer and calling SWI OS_Memory to fill in the corresponding physical addresses.

2 On a DMA transfer from device to memory the DMA manager asks the memory manager to mark pages as uncacheable so that reads from the pages being DMAed into return the transferred data and not cached data. This is done in combination with the SWI OS_Memory call. The cache must also be flushed before the transfer is started.

3 The memory manager broadcasts Service_PagesUnsafe when it is about to remap some physical pages (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 must scan its page tables for all active transfers and temporarily halt 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 can then continue any halted transfers using the new physical addresses.

See the chapter entitled *Memory management* on page 9 for more details.

Programmer interface

Device drivers

Device drivers register with the DMA manager which logical channels (devices) they control. The device drivers then queue DMA requests which the DMA manager processes in order.

SWIs used for DMA control

The following SWI calls are used for DMA control:

DMA_RegisterChannel	registers a client device with the DMA manager
DMA_DeregisterChannel	deregisters a client device with the DMA manager
DMA_QueueTransfer	queues a DMA transfer request
DMA_TerminateTransfer	terminates a DMA transfer request
DMA_SuspendTransfer	pauses an active DMA transfer
DMA_ResumeTransfer	restarts a paused DMA transfer
DMA_ExamineTransfer	return the progress of an active DMA transfer

These SWI calls are described in detail starting on page 64.

Control routines

A device driver calls the SWI DMA_RegisterChannel to register itself as the controller of the specified logical channel. The value passed in R4 is a pointer to a word aligned table of control routine addresses.

The vector of control routines registered with the DMA manager (using SWI DMA_RegisterChannel) by a device driver will be called as follows for a normal transfer.

Start
Enable DMA
    <transfers>
        ....
        ....
    <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 and interrupt status as necessary but must return to the DMA manager by using the instruction MOVS r15, r14 (or equivalent LDM).This restores the processor mode, interrupt status and flags 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 in order to return an error.Note that enabling interrupts in callback routines may have undesirable effects and should be avoided.Note also that calling DMA manager SWIs from these callback routines is not advised and may cause problems.

Turn to SWI DMA_RegisterChannel on page 64 for more information on their use.
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

Interrupts

Interrupt status is undefined
Fast interrupt status is undefined

Processor mode

Processor is in IRQ or SVC mode

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

Interrupts

Interrupt status is undefined
Fast interrupt status is undefined

Processor mode:

Processor is in IRQ or SVC mode

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

Interrupts

Interrupt status is undefined
Fast interrupt status is undefined

Processor mode

Processor is in IRQ or SVC mode

Use

The DMA manager calls this control routine before starting a new DMA request. This call is only be 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' routine is then called with the same error. If this call returns with no error then the DMA manager then calls the 'Enable DMA' control routine.

Completed

On entry

R0 = 0 (Vclear) or points to error block (Vset)
R11 = R2 from DMA_QueueTransfer call
R12 = R5 from DMA_RegisterChannel call

On exit

All registers preserved

Interrupts

Interrupt status is undefined
Fast interrupt status is undefined

Processor mode

Processor is in IRQ or SVC mode

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'
'Insufficient memory pages available'
'No readable memory at this address'

DMASync

On entry

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

On exit

R0 = 0 for continue
= N for stop after *n* bytes
All other registers preserved

Interrupts

Interrupt status is undefined
Fast interrupt status is undefined

Processor mode

Processor is in IRQ or SVC mode

Use

This is provided for real-time synchronisation with DMA transfers. This is essential for time critical device drivers where the driver has to know how far a transfer has progressed. This callback is only made if bit 2 of R0 was set in the DMA_QueueTransfer call and it is made after every transfer of the number of bytes specified in R6 in the same DMA_QueueTransfer call. 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*s+t bytes where s=gap between DMASync calls and

t=transfer unit size. 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 with the DMA manager.

On entry

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

On exit

R0 = channel registration handle
All other registers preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

A device driver calls this SWI to register itself as the controller of the specified logical channel. 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. See the section entitled *Control routines* on page 59 for a full description of their use.

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

Related SWIs

DMA_DeregisterChannel (SWI &46141)

Deregisters a device previously registered with DMA_RegisterChannel

On entry

R0 = channel registration handle

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 not re-entrant

Use

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

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

DMA_QueueTransfer (SWI &46142)

Queues a DMA transfer for a logical channel

On entry

R0 = flags

bit	meaning
0	transfer direction 0 = from device to memory (read) 1 = from memory to device (write)
1	1 = scatter list is a circular buffer
2	1 = call 'DMASync' callback
3 – 31	reserved (must be set to 0)

R1 = channel registration handle
R2 = value of R11 to be passed to control routines
R3 = address of scatter list (word aligned)
R4 = number of bytes to transfer,
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 'DMA sync' callbacks 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 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 should not be remapped by the client 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 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 SWI DMA_TerminateTransfer or by the DMASync callback.

The value passed in R4 determines the number of bytes to be transferred. If the transfer uses a circular buffer then this value can be 0 to indicate an infinite length transfer. This value must be a multiple of the transfer unit size, and if a circular buffer is not used then it 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 drivers 'DMASync' callback routine. This is provided so that real-time synchronisation is possible for certain device drivers (for example Sound). The value in R6 must be a multiple of the transfer unit size.

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

DMA_TerminateTransfer (SWI &46143)

Terminates a DMA transfer request

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 the request queued by DMA_QueueTransfer

If the DMA transfer is active then it is stopped and the DMA manager calls the Disable DMA control routine. Otherwise, the request is simply removed from its queue. The DMA manager will always call the 'Completed' control routine (on page 61) with V set and R0 pointing to the supplied error block.

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

An error is returned if the DMA tag is invalid.

DMA_SuspendTransfer (SWI &46144)

Suspends an active DMA transfer

On entry

R0 = flags

bit	meaning
0	0 ⇒ don't start queued transfers, 1 ⇒ start next queued transfer
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, 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.

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 when the suspended transfer is restarted.

An error is returned if the DMA tag is invalid.

DMA_ExamineTransfer (SWI &46146)

Returns the progress of an active 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 the total number of bytes transferred by the specified active DMA operation. An error is returned if the DMA tag is invalid, or the specified DMA operation is not active.

5 Parallel and serial device drivers

Introduction

This chapter outlines the changes made to the serial and parallel device drivers in order to speed up the performance of these ports.

The parallel device driver now uses the new buffer manager interface. The parallel device can be opened either for input or output but not for both. A new device provides a fast Centronics mode where data is automatically sent by the hardware at a very high transfer rate.

The serial device driver performance has been improved. It can now support a maximum serial port rate to 115200 baud. Additionally, interrupt problems affecting serial input have been virtually eliminated.

Overview

Buffer manager

The buffer manager has been extended to provide an interface for inserting and removing data to and from buffers without the need to use SWI calls (with all of the related overheads). This interface takes the form of a service routine which can be called directly and which performs a variety of functions. A new SWI has been provided by the buffer manager SWI Buffer_InternalInfo (SWI &42949).

Buffer manager service routine

The buffer manager service routine is passed an internal buffer ID, so that the buffer manager can go straight to the appropriate buffer record in its workspace rather than performing a linear search on the buffer handle. 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. 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 use the new buffer manager interface in all situations where InsV, RemV or CnpV are used. For example, in the filing system interface, DeviceFS_ReceiveCharacter and DeviceFS_TransmitCharacter.

Technical Details

Parallel device driver

The parallel device driver has been modified to use the new buffer manager interface. 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 SWI Buffer_InternalInfo 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 are 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 using STROBE and BUSY signals as the handshake. The parallel device driver accesses this mode using a new device called 'fastparallel:'. The 'parallel:' device is still available as the default as some printers cannot cope with the fast transfer rate of the new device. To cope with fast parallel the printer must assert BUSY within 500ns of receiving STROBE.

Serial device driver

The serial device driver does not use the buffer manager interface; this is to retain maximum compatibility with existing applications that use the serial interface.

Performance improvements have increased the maximum serial port rate to 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.

Baud rates

An additional number of higher baud rates are selectable. The configurable baud rates are:

Value	Baud rate
0	9600
1	75
2	150
3	300
4	1200
5	2400
6	4800
7	9600
8	19200
9	50
10	110
11	134.5
12	600
13	1800
14	3600
15	7200
16	38400 (new)
17	57600 (new)
18	115200 (new)

These are configured and selected using the existing interfaces.

I/O chip type

The type of I/O chip present and what features it has can be determined with the updated version of OS_ReadSysInfo (SWI &58). This SWI is described in the RISC OS 3 *Programmer's Reference Manual* on *page* 1-720. Only changes to this interface description are explained here.

OS_ReadSysInfo 2 (SWI &58)

On entry

R0 = 2 (reason code)

On exit

R0 = hardware configuration word 0
- bits 0 - 7 = special functions chip type
  - 0 ⇒ none
  - 1 ⇒ IOEB
- 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'655 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)

R3 = word 0 of unique machine ID, or 0 if unavailable
R4 = word 1 of unique machine ID, or 0 if unavailable

OS_ReadSysInfo3 (SWI &58)

On entry

R0 = 3 (reason code)

On exit

R0 = I/O chip basic features mask
R1 = I/O chip extra features mask
R2 - R4 reserved for future expansion

Use

Unaltered except for the values returned for the chip types.

R0 bits	sub-unit	710	711	SMC 655
0 - 3	IDE	1	1	1
4 - 7	floppy	1	1	1
8 - 11	parallel	1	1	1
12 - 15	1st serial	1	1	1
16 - 19	2nd serial	0	1	1
20 - 23	config	1	2	3
24- 31	reserved	0	0	0
R1 bits	**sub-unit**	**710**	**711**	**SMC 655**
0 - 3	IDE	0	0	0
4 - 7	floppy	0	0	0
8 - 11	parallel	0	0	1
12 - 15	1st serial	0	0	1
16 - 19	2nd serial	0	0	1
20 - 23	config	0	0	0
24- 31	reserved	0	0	0

OS_ReadSysInfo4 (SWI &58)

On entry

R0 = 4 (reason code)

On exit

R0 = LSW of Ethernet Network Address (or 0)
R1 = MSW of Ethernet Network Address (or 0)

Use

Code loaded from the dedicated Network Expansion Card or from a normal expansion card should use the value returned by this call in preference to a locally provided value.

Sample code

```
GetNetAddress                           ; Return in R0 and R1
        STMFD   sp!, {lr}
        MOV     r0, #4
        SWI     XOS_ReadSysInfo
        BVS     TryLocally
        ORRS    r14, r0, r1             ; Test for both zero
        LDMNEFD sp!, {pc} ^             ; Return if OK
TryLocally
        STMFD   sp!, {r0-r4}
        LDR     r3, MyHardwareAddress
        MOV     r0, #0
Loop
        SWI     XPodule_EnumerateChunks
        BVS     Exit
        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
Exit
        STRVS   r0, [ sp, #0 ]
        LDMFD   sp!, {r0-r4, pc}
ErrorExit
        ADR     r0, Error
        CMP     pc, #&80000000          ; Set V
        B       Exit
Error
        DCD     ErrorNumber_NoNetworkAddress
        DCB     "Unable to find an address for Ethernet", 0
        ALIGN
```

OS_ReadSysInfo 2 provides information on the general hardware configuration of the machine, including memory and video control. These have been included for clients such as the new ScreenModes module which must determine whether VIDC20 is present. Note that on existing hardware OS_ReadSysInfo 2 returns 0 or 1 in R0 which is compatible with the above interface changes. The parallel device driver uses this call to determine whether IOMD is present so that the parallel interrupt can be cleared in IOMD or in an external latch as at present.

The values returned in R0 by OS_ReadSysInfo 3 only differ in that the configuration for the SMC'665 is different to the 82C711. The sub-units described still have the same basic functionality. However, it is possible with the SMC'665 to use a fast parallel mode (with FIFO and hardware handshake) and to use the serial FIFOs provided. Hence the extra features mask returned in R1 updated to reflect the additional functionality.

OS_ReadSysInfo 4 is a new call which returns the Ethernet Network Address if available.

New low-level serial operations

The SWI OS_SerialOp (SWI &57) has been extended to allow serial speeds to be enumerated more easily and to set the serial hardshake extent.

OS_SerialOp 7 (SWI &57)

Emulate the control interface of the old 6850 controller used in the BBC B.

This SWI is not to be used. It is for internal use only.

OS_SerialOp 8 (SWI &57)

Set serial handshake extent

On entry

R0 = 7 (reason code)
R1 = -1 to be read or new value to write

On exit

R0 preserved
R1 = old value

Use

This reason code provides the same functionality as the old OS_Byte 203 call used to set the serial threshold value.

OS_SerialOp 9 (SWI &57)

Enumerate serial 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

Use

The table returned is word aligned and each word in the table specifies a supported baud rate in 0.5 bit/sec units.

Baud rate = table entry / 2

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 & 6 calls to set the corresponding baud rate.

Buffer manager service routine

This service routine is passed an internal buffer ID so that the buffer manager can go to the appropriate buffer record in its workspace. These are used by the SWI Buffer_InternalInfo described on page 88.

This SWI converts the buffer handle passed in R0 to a buffer manager internal buffer ID. The address of the buffer manager service routine and a value to quote in R12 when the service routine is called is also returned.

A device driver should create or register its buffers with the buffer manager and then call this SWI to obtain the internal buffer ID for each buffer and store the service routine address and R12 value in its workspace.

When a buffer is removed or deregistered it is the device driver's responsibility to ensure that it no longer calls the buffer manager service routine with the internal buffer id for that buffer.

If the buffer handle is invalid an error is returned but can be ignored. The service routine address and R12 value are always returned.

Reason codes

On entry

R0 = reason code (see below)
Other registers depend on reason code

On exit

Other registers depend on reason code

Interrupts

Interrupts may be enabled or disabled

Processor mode

IRQ or SVC mode

Use

The buffer manager service routine is passed an internal buffer ID so that the buffer manager can go straight to the appropriate buffer record in its workspace rather than performing a linear search on the buffer handle. 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. The existing vector interface is still supported but takes the form of an extra layer on top of the new code.

The particular action of OS_File is given by the low byte of the reason code in R0 as follows:

R0	Action	Page no.
0	Insert byte	
1	Insert block	
2	Remove byte	
3	Remove block	
4	Examine byte	
5	Examine block	
6	Return used space	
7	Return free space	
8	Purge buffer	
9	Next filled block	

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 ⇒ failed to insert

Use

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 ⇒ unable to transfer all data (ie. R3≠0)

Use

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 been written directly into the buffer ie. R2 = 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

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

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

Returns 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

Allows a block of data in the specified buffer to be examined 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

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

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 other registers preserved

Use

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 ⟹ buffer empty

Use

This call can be used to remove buffered data directly rather than copying the data 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.

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.

SWI calls

Buffer_Internalinfo (SWI &42949)

Converts the buffer handle passed in R0 to a buffer manager internal buffer I.D.

On entry

R0 = buffer handle
R1 = file handle

On exit

R0 = internal buffer ID
R1 = address of buffer manager service routine
R2 = value to pass to service routine in R12

Interrupts

Interrupts are not affected
Fast interrupts are not affected

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. The address of the buffer manager service routine and a value to quote in R12 when the service routine is called is also returned. The service routine address and R12 value are always returned. The buffer manager service routines are described on page 81.

A device driver should create or register its buffers with the buffer manager and then call this SWI to obtain the internal buffer ID for each buffer and store the service routine address and R12 value in its workspace.

When a buffer is removed or deregistered it is the device driver's responsibility to ensure that it no longer calls the buffer manager service routine with the internal buffer id for that buffer.

If the buffer handle is invalid an error is returned but can be ignored.

6 Video

Introduction

The video system in the new architecture has been substantially changed so that the new video controller chip, the VIDC20, can be used to its full capabilities.

Using VIDC20 gives a much improved video capability over the previous generation of computers that used VIDC1 or VIDC1a chips.

Colours on the desktop

The following colour options are supported on the desktop:

- 1 bpp (bit per pixel)
- 2 bpp
- 4 bpp
- 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 and resolutions

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 also a trade-off between the chosen resolution and the processor bandwidth used for video and computer usage.

Screen resolutions

The following maximum screen resolutions are supported on the desktop:

Using DRAM as screen memory

1024 X 768	4 bpp
800 X 600	8 bpp
aaa X bbb	16 bpp
aaa X bbb	32 bpp

Using 1MB of VRAM as screen memory

aaa X bbb	4 bpp
1024 X 768	8 bpp
800 X 600	16 bpp
aaa X bbb	32 bpp

Using 2MB of VRAM as screen memory

aaa X bbb	4 bpp
aaa X bbb	8 bpp
1024 X 768	16 bpp
800 X 600	32 bpp

These are only the maximum limits that can be used. The Display manager utility allows a selection of pre-defined modes to be chosen; custom modes can also be used and defined.

Terminology

The following terminology is used throughout this chapter:

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 always has bit0 of its flags word set, so it can be distinguished from a sprite area.

Some calls use a *mode specifier*. A mode specifier can be an old-type mode number or a pointer to a mode selector.

In addition some Wimp calls take a simplified string, the *mode string*, that defines the display mode.

The sprite mode word is now termed the *Sprite Type*; it defines the fields necessary for the display mode in terms of resolution and colour. It is stored as part of the new sprite format header (see below for more information).

Other features

Identifying monitor types

The monitor lead identification system used to identify monitors is only used to distinguish between standard and VGA type monitors. A new module called ScreenModes is used to identify monitor types and to read an exact monitor specification from disc. This allows monitors to be set up accurately and correctly.

Word-wide interfaces

There are new SWI interfaces to replace interfaces where a single byte is used to specify a colour. Other modification have been made to existing interfaces to make them word-wide.

Colour-matching algorithms

ColourTrans now works with the much larger number of colours available.

ColourTrans

To support VIDC20, the ColourTrans code has been modified. ColourTrans_SetColour and ColourTrans_SetOppColour have now been changed so that OS_SetColour implements these calls.

The SWI call ColourTrans_SelectTable now handles the new 16bpp and 32bpp modes. These colours can be returned as half words or words rather than bytes.

ColourTrans has been extended to support the new colour modes. Several ColourTrans SWIs have been modified to handle the new 16bpp and 32bpp colour modes. Colours can now be returned as words rather than bytes.

Where ColourTrans can be passed a mode number, it can now also be passed a *mode selector*. A mode selector defines one of the new types of mode.

8bpp model retained

The existing 8bpp VIDC1 model of 64 colours and 4 tints is retained in 256 colour modes with the default palette. If specifically needed a free-hand palette of 256 different colours can also be setup.

VDU calls extended

Many VDU SWI calls now accept a mode specifier in R0, not just a mode number. There are also some VDU calls should not be used in 16bpp and 32bpp display modes.

In addition several VDU service calls have been extended, in particular Service_ModeExtension (Service call &50) now supports a type 3 register format list. This is a new format VIDC list is used in RISC OS which is independent of the video controller used. For more information see page 113.

A new service call Service_EnumerateScreenModes has been created to supply applications that choose display modes with information about the pixel depths available.

WIMP

The call Wimp_SetMode (SWI &400E3) previously took a mode number, it now also accepts a mode specifier.

The command *WimpMode can now use the mode string to specify the screen display. It allows the mode to be specified either as a number, or in the form of a mode description string, which is a textual form of a mode selector. This is also reflected in the Display manager application, which also allows this form. The format of the mode description string is described on page 98.

Sprite format

The support for new colour modes – particularly the 16bpp and 32bpp display modes – has meant that a new sprite type format had to be created. This is now used with all 16bpp and 32bpp sprites. This new sprite type contains a new sprite format header to distinguish it from an existing sprite type. The new sprite format header contains the Sprite type (formally the sprite mode word in RISC OS 3.1).

Overview

ColourTrans colour matching facilities

A new colour matching algorithm has been implemented in ColourTrans. This allows it to deal with the much higher number of colour matches needed by 8bpp modes with independent palette entries.

The output of the algorithm is a 32K table containing colour matches for 5bits each of Red, Green and Blue (for 16bpp or 32bpp mapping down to 8bpp or below).

This table is generated when it is needed; it takes under 2 seconds to generate. In regularly used cases a precalculated table is used. For example, a precalculated table is used when 16bpp and 32bpp are mapped to 8bpp (VIDC1 palette) and when 16bpp and 32bpp are mapped to 8bpp greyscale (assuming the default 8bpp palette in each case).

8bpp takes longer to calculate but there are precalculated tables so there is normally no calculation time at all.

The address of the colour match table is exported through ColourTrans_SelectTable and ColourTrans_GenerateTable whenever mapping from 16bpp and 32bpp down to 8bpp or below. The table remains valid until the next palette change, mode change, or switch of output to sprite/screen.

The call only generates the table when a new table needs to be built; when an application is in doubt whether the current table is correct it must call ColourTrans again. The call either returns immediately because the table is still correct or a precalculated table is available or it builds a new table or points at a precalculated table and returns.

When an old format sprite with a full palette (256 pairs of entries for an 8bpp sprite) is plotted in 16bpp or 32bpp, SpriteExtend ignores any translation table provided and uses the RGB values contained in the palette to plot the sprite.

RISC OS 3.X sprite format

The support for new colour modes – particularly the 16bpp and 32bpp display modes – means that a new sprite format has been created.

This new sprite format avoids the problems caused by binding sprite files to a mode number not available on the viewing computer. For example, by creating a sprite that specified a soft-loaded screen mode.

Where a suitable screen mode exists, sprites are saved in the old format to retain compatibility with RISC OS 3.1 computers. 16bpp and 32bpp sprites are always saved in the new sprite format whilst 1bpp, 2bpp, 4bpp and 8bpp sprites are saved in the old sprite format (if possible).

Screen modes

Because of the increased number of colours and the range of resolutions available using VIDC20, the way of choosing colours and resolutions using the mode number interface became limiting (there was a maximum of only 128 modes with only 64 available to Acorn). To bypass this limitation a new way of selecting screen displays is now used.

The RISC OS mode extension system has also been extended and new list format has been created to take advantage of VIDC20 capabilities.

New module ScreenModes

VIDC20 supports a much wider range of monitors than did VIDC1. The range of line frequencies available on monitors varies widely, rather than create a many new monitor type numbers, a new ScreenModes module is used to read the timings for the full set of screen modes from monitor definition files on the hard disc.

These monitor definition files contain a specific monitor description file that sets up accurate monitor timings. There is a separate file for each supported monitor.

These files (called ModeInfo files) are loaded into memory using the *command *LodeModeFile, if the file contains valid information, it sets the current monitortype to 7 (file) (using OS_ScreenMode(R=3)). This then makes available all the screen modes defined in the file, while removing all modes defined in any previously loaded file. *LoadModeFile is defined on page XX, while OS_ScreenMode is defined on page XX.

Additional information

In RISC OS 3.1 screen modes were selected by sending the VDU sequence (22, n) where n is the screen mode number. The value of n is limited to 8 bits and bit 7 is used as a shadow mode indicator; there are in fact only 128 screen modes available using this interface. Acorn allocates 64 of these as available for use by third parties, so that leaves only 64 for use by Acorn.

The RISC OS mode extension system has also been extended. In RISCS OS 2 and RISC OS 3.1 modules that provide extra screen modes currently respond to services, returning lists which contain values to be programmed into VIDC1. A new list format is necessary to take advantage of VIDC20 capabilities. Old-style formats are not supported.

Support for external video cards

No additional support is provided in RISC OS for video expansion cards. Existing interfaces provide enough hooks for these to be added without additional support.

An external video card should redirect the video bitmap to an area of RAM on an expansion card. This would be in EASI space, and so it can support full 32-bit access.

RISC OS screen addresses are then changed to point to EASI space. The mechanism used to change screen address, switches VDU output into a sprite, where the sprite is situated in EASI space. This requires some RAM immediately before the start of the bitmap, to hold the sprite area and sprite headers.

The mouse pointer position can be polled by the external card drivers, then the hardware on the card producing the screen pointer can be updated with the pointer's new position.

User interface

Display manager

Mode changes are handled by a new application, the Display manager.

The Display manager makes the choice of screen mode easy for ordinary users. The old mode number scheme was complicated and difficult to understand.

With the Display manager, screen modes are described using the number of colours provided and the resolution of the screen. These may be linked with terms such as VGA and SVGA. For more information see the chapter entitled *User interfaces* on page 293

Defining the monitor type

Screen Modes are selected using the Display manager. However, before selecting your Mode, you need to define the monitor type you are using.

The monitor types available have been expanded; there are now six monitor types and an auto monitor type. For more information chapter entitled *User interfaces* on page 293

Technical details

The mode string syntax

The mode string definition is used to define a particular screen display. It is used by several Wimp calls and the command *WimpMode. It is also used by the Display manager utility.

It uses 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, 16, 64, 256, 32T, 32K, 16M)
G*ggg*	Greys (*ggg* = 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)

Some examples:

X640 Y512 C16	mode 20
X640 Y480 C16 EX0 EY0	mode 27 with extra-large text
X320,Y480,C64	VIDC 1 style 8bpp, VGA with rectangular pixels

If you want to use one of the old type modes, as defined by a mode number, you can enter a mode number (into the Display manager utility) rather than a mode string. For example you can put in 15 to define the old type mode 15.

- 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, and if left out causes -1 to be used instead.

The Display manager utility only changes modes using *Wimpmode. A mode selection string is constructed when the user clicks on OK in the window.

The mode selector format

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 always has bit0 of its flags word set, so it can be distinguished from a sprite area.

Some calls use a *mode specifier*. A mode specifier can be an old-type mode number or a pointer to a mode selector.

A mode selector is a word-aligned structure of the following format:

Offset	Value
0	mode selector flags bit 0 = 1 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 means 'use first match')
20	pairs of words (mode variable index, value – 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 XX for more information.

Passing mode selectors to ColourTrans

Where ColourTrans can be passed a mode number it also accepts a mode selector. A mode selector always has bit0 of its flags word set, so it can be distinguished from a sprite area. The first word of a sprite area is the size of the area – which must be word aligned, so bit0 will always be 0.

The mode specifier format

Where a call takes a mode specifier, it is either a mode number (in the range 0 to 255) or a pointer to a mode selector (greater than 255). The range of the value determines which. However, in most cases you may also use a Sprite Mode Word too.

The sprite header format

The new sprite type avoids the problems caused by binding sprite files to a mode number not available on the viewing computer.

Sprite header format (new type)

Word	Meaning
1	Offset to next sprite
2 – 4	Sprite name
5	Width in words –1
6	Height in lines –1
7	0 (reserved for future use) ** No left hand wastage is allowed on new format sprites.
8	Last bit used (right end of row)
9	Offset to sprite image
10	Offset to 1bpp mask/image **
11	New Sprite Mode Word **
12	Palette data (if present) or start of image

By defininition there is no left hand wastage.

** These words have changed.

The New Sprite Mode Word

The New Sprite Mode Word uses the meaning of bits 27 to 31 to define the sprite type.

When the Type is non-zero it has the following fields:.

Bit	Mealng
0	1 – distinguishes it as a sprite type rather than a mode selector to OS_ReadModeVariable and ColourTrans
1 to 13	Hdpi
14 to 26	Vdpi
27 to 31	T (the Type field)

The meaning of bits 27 to 31 are as follows.

T value	Meaning
0	This is the backward compatibility mode.†
1	1bpp. ‡ New mask format (same as old mask)
2	2bpp. ‡ New mask format
3	4bpp. ‡ New mask format
4	8bpp. ‡ New mask format
5	16bpp. ‡ New mask format, other as T=1.
6	32bpp. ‡ New mask format, other as T=1.
7	CMYK. Not supported within RISC OS
8	24bpp. Not supported within RISC OS
9	JPEG
10 to 31	For future expansion.

† When the type is zero, it is an old format mode word, so bits 00 to 26 are the Mode number. The mode is in bits 0 to 6.

‡ No palette. [also applies to 16bpp and 32bpp]
Vdpi and Hdpi are Vertical and Horizontal dots per inch of the sprite. The only recommended Xdp x Ydpi are 90 x 90 and 90 x 45, which correspond to square and rectangular pixels. For example a 16 byte palette is two 8 bit palette entries. This leaves room in the specification for more efficient palettes.
If present, mask data is 1bpp regardless of image depth.

Notes on Pixel depth/data table values

Zero (0) gives old format without changes. This is the only format in which an attached palette is supported. For valuse of T which are greater than 0 the palette is a new format (which is not currently implemented).

Pixel storage for 16/32bpp is:

16bpp

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

32bpp

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

When a new sprite format is forced back to a mode number the following logic applies.

Xdpi	Ydpi	bpp	Mode number
90	45	1	0
90	45	2	8
90	45	4	12
90	45	8	15
45	45	1	4
45	45	2	1
45	45	4	9
45	45	8	13
90	90	1	25
90	90	2	26
90	90	4	27
90	90	8	28

- 16bpp and 32bpp sprites are always in the new format.
- See the RISC OS 3 *Programmer's Reference Manual* for more information about mode choices.

The sprite type and OS_SpriteOp

The SWI OS_SpriteOp supports the following sprite types. The definitions of sprite type T is given on page 100. Type 0 is the compatibility mode, and Type 5 and Type 6 correspond to 16bpp and 32bpp modes:

Sprite Op	Use	Sprite type T
2	Screen save	0, 1 to 6 no palette
3	Screen load	0, 1 to 6 no palette
8	Read area control block	–
9	Init sprite area	–
10	Load sprite file	–

Sprite Op	Use	Sprite type T
11	Merge sprite file	–
12	Save sprite file	–
13	Return name	0 to 6
14	Get sprite	0, 5 and 6 (1 to 4 forced to 0 in old modes)
15	Create sprite	0, 5 and 6 (1 to 4 forced to 0 in old modes)
16	Get use sprite co-ords	0, 5 and 6 (1 to 4 forced to 0 in old modes)
24	Select sprite	–
25	Delete sprite	–
26	Rename	–
27	Copy sprite	–
28	Put sprite	0, 1 to 6 no palette
29	Create mask	0, 1 to 6 no palette
30	Remove mask	0, 1 to 6 no palette
31	Insert row	0, 1 to 6 no mask/palette
32	Delete row	0, 1 to 6 no mask/palette
33	X axis flip	0, 1 to 6 no mask/palette
34	Put sprite user co-ords	0, 1 to 6 no palette
35	Append sprite	0, 1 to 6 no mask/palette
36	Set pointer shape	0 to 4 no mask/palette
37	Create/remove palette	0 to 6 (can only create for T = 0)
40	Read sprite info	0 to 6
41	Read pixel colour	0, 1 to 6 no palette
42	Write pixel colour	0, 1 to 6 no palette
43	Read pixel mask	0, 1 to 6 no palette
44	Write pixel colour	0, 1 to 6 no palette
45	Insert column	0, 1 to 6 no mask/palette
46	Delete column	0, 1 to 6 no mask/palette
47	Y-axis flip	0, 1 to 6 no mask/palette
48	Plot mask	0, 1 to 6 no palette
49	Plot mask user co-ords	0, 1 to 6 no palette
50	Plot mask scaled	0, 1 to 6 no palette
51	Paint char scaled	–

Sprite Op	Use	Sprite type T
52	Put sprite scaled	0, 1 to 6 no palette
53	Put sprite grey scaled	0 (with given restrictions)
54	Remove LH wastage	0, 1 to 6 no mask/palette
55	Plot mask transformed	0, 1 to 6 no palette
56	Put sprite transformed	0, 1 to 6 no palette
57	Insert/delete rows	0, 1 to 6 no mask/palette
58	Insert/delete columns	0, 1 to 6 no mask/palette
60	Switch out to sprite	0, 1 to 6 no palette
61	Switch out to mask	0, 1 to 6 no palette
62	Read save area size	-
64	Reserved, not implemented	
65	Reserved, not implemented	

Mask data structure

Whatever the depth of image, the mask 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.

The sprite type and OS_ReadModeVariable

Values returned for relevant variable numbers are shown below. These show the new values returned for the new sprite type values:

VDU variable number	Name		
	NColour		
		T	Returned value
		1	1
		2	3
		3	15
		4	255
		5	65535
		6	2^32-1
4	XEigFactor		
5	YEigFactor		
		ppi	EIG
		22/23	3
		45	2
		90	1
		180	0
9	Log2BPP		
10	Log2BPC		
		T	Log2BPP/ Log2BPC	BPP=BPC=
		1	0	1
		2	1	2
		3	2	4
		4	3	8
		5	4	16
		6	5	32

NColour

In the SWI OS_ReadModeVariable, Ncolour=63 is returned for all 8bpp VIDC1 screen modes. Ncolour=255 is returned for 8bpp non-VIDC1 screen modes; i.e. full palette.

Modeflags

In the SWI OS_ReadModeVariable, Modeflags bit 7 is set for 8bpp non-VIDC1 screen modes.

Monitor description information 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 followed by the hash character (#) are treated as comments and ignored.

The file consists of the following two lines:

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

followed by any number of mode definitions, where

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

and a mode definition is as follows:

```
startmode
        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
        mode_name: mode name
endmode
```

where

x-resolution	is the number of pixels displayed across the screen
y-resolution	is the number of displayed rasters

The h_timings line controls the horizontal timings in units of pixels, as follows:

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 (normally the same as *x-resolution*)
hrbdr	is the width of the right hand border
hfpch	is the width of the horizontal front porch

The v_timings line controls the vertical timings in units of raster lines, as follows:

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 required, 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
4	composite syncs

mode name	is a textual name for the mode for use in menus and such. The mode name field must be present, although the <mode name> itself may be blank.

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.

ColourTrans

GCOL compatibility

In 16bpp and 32bpp modes, 8 bit GCOL assignments made via VDU16 and VDU17 work as if in an 8bpp mode.

ColourTrans GCOL calls such as ColourTrans_ReturnGCOL and ColourTrans_SetGCOL accept a word value for 16bpp and 32bpp.

ColourTrans_SelectTable (SWI &40740)

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.

Mapping between modes

The table size generated by an application attempting to map down from a 16 or 32bpp to a 1-8bpp mode is excessively large, so ColourTrans does not return full translation tables in these cases.

The revised ColourTrans_SelectTable functionality is:

		Source Mode	
		1, 2, 4, 8 bpp	16, 32 bpp
Destination Mode	1, 2, 4, 8, bpp	1	2
	16, 32 bpp	3	4

Key:

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. When the table is first calculated the call may take a few seconds to return. This table is only valid until the next palette change, mode change or switch output to screen/sprite. The structure 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.

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.

A new flag, bit 4 of R5, instructs the call to return >8 bits per colour rather than bytes (indicating that the caller is aware that the colours/bytes relationship no longer holds true).

```
R5 = flags
        bit 4 set ⇒return > 8bits 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.

Supremacy bits

The SWIs ColourTrans_ReadPalette (SWI &4075C) and ColourTrans_WritePalette (SWI &4075D), 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 four of these bits are used, bits 7 to 4. In other modes one bit is used, bit 7.

The palette entry passed through these calls is in the form &bbggrrs0, where s is the supremacy mask nibble.

Where there is only one bit of supremacy it appears in bit 7. Where there are four bit they appear in bits 7 - 4. ColourTrans and the kernel now support this (the kernel only expects one bit of supremacy and ignores the rest).

SpriteOps for calibrated plotting of 16bpp and 32bpp

Two new sprite operations are being defined but not implemented (other than reserving their numbers). Both are included to allow 16/32bpp sprites to be plotted with calibration. It is realised that this will be extremely slow. The exact internal operation of these calls is to be defined in the future.

Applications wishing to use these calls should call the new numbers, get an error, and then use an existing uncalibrated call instead.

Parameters for both calls are as per the uncalibrated version.

63 PutSpriteScaledCalibrated
64 PutSpriteTransformedCalibrated

Grey level modes

The grey-level mode is just like a colour mode with the same pixel depth; the palette is initialised just to provide grey levels. The kernel does not know specifically about grey-level modes.

The Window Manager now allows the selection of 16 and 256 level grey scale modes in the desktop. The Wimp explicitly programs the palette after the mode change. 256-colour modes are selected with a fully programmable palette using the SWI OS_ScreenMode.

The 16 grey-level palette is set by calling Wimp_SetPalette. The first eight desktop 'colours' stay the same and the next eight provide interpolated greys. This means that the logical colours do not decrease in brightness monotonically.

Gamma correction

RISC OS now allows an application to supply 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 VIDC20, 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. In a similar fashion the green and blue components are looked up in their respective tables.

The contents of the tables are set up by a call to PaletteV Reason code 9. See page 116 for a full description of the revised PaletteV call.

ColourTrans SWIs

These SWIs now accept mode specifiers rather than mode numbers:

ColourTrans_ReturnGCOLForMode (SWI &40745)
ColourTrans_ReturnColourNumberForMode (SWI &40746)
ColourTrans_ReturnOppGCOLForMode (SWI &4074A)
ColourTrans_ReturnOppColourNumberForMode (SWI &4074B)
ColourTrans_SelectTable (SWI &40740)
ColourTrans_SelectGCOLTable (SWI &40741)
ColourTrans_ReadPalette (SWI &4075C)
ColourTrans_GenerateTable (SWI &40763)

The sprite area pointer case was distinguished by the value being greater than or equal to 256 (since mode numbers are only byte quantities).

A pointer to a mode selector (mode specifier) is distinguished from a pointer to a sprite area because the first word of a sprite area contains the size of the sprite area, which must be a whole number of words, therefore bits 0 and 1 will be clear. The first word of a mode selector is the flags word which always has bit 0 set.

VDU – extended SWIs

OS_SetColour (SWI &61)

Two new flags have been added:

R0 bit 6, to set the text colour.
R0 bit 7 to permit the colour setting to be read for restoration later.

On entry

R0 = flags:

bits 0 - 3	graphics plotting action (see below)
bit 4	set ⇒ alter background, clear ⇒ alter foreground
bit 5	set ⇒ R1 = pattern data, clear ⇒ R1 = colour number
bit 6	set ⇒ R1 = text colour, clear ⇒ R1 = graphics colour
bit 7	set ⇒ read colour, clear ⇒ set colour
bit 8 to 31	reserved – set to 0

R1 = If R0 is bit 5	colour number (if R0 bit 5 is clear) or pointer to eight words of pattern data (if R0 bit 5 is set)
If R0 is bit 6	pointer to pattern block (if R0 bit 6 is clear)

On exit

Set	all registers preserved
Read	R0 = flags
	R1 = colour number or pointer to pattern block

Use for this computer

When setting the colour all the flags are used. When reading the colour only the foreground/background flag and the text colour flag is used. A pattern block must be supplied for reading as this will be filled in with the ECF if necessary. The values returned by reading the colour are ready for passing straight to OS_SetColour to set the colour back. Reading returns the following:

Graphics colour wanted:

R0 returns:

bit	meaning
0 to 3	logical operation
4	preserved (fg/bg flag)
5	1 (pattern block flag)
6	0 (text/graphics flag)
7	0 (read/write flag)
8 to 31	preserved

R1 unchanged, pattern block filled in

Text colour wanted:

R0 returns:

bit	meaning
0 to 3	logical operation
4	preserved (fg/bg flag)
5	0 (pattern block flag)

6	1 (text/graphics flag)
7	0 (read/write flag)
8 to 31	preserved

R1 colour number

OS_ReadModeVariable (SWI &35)

In RISC OS 3.1 this call is passed in R0 either the mode number, or –1, indicating the current screen mode. For this computer, R0 has now been extended so that:

R0 = mode number or a mode specifier (pointer to mode selector) or a new format sprite mode word or –1 (indicating the current screen mode).

OS_ReadModeVariable distinguishes between the three mode values as follows:

0 ≤ R0 ≤ 255	Mode number
R0 bit 1 set	New format sprite mode word
R0 is a pointer to a word aligned mode selector	Mode selector

Bit 0 of the new format sprite word is always set. Bit 0 of the pointer to mode selector must be clear; mode selectors always start on a word boundary.

OS_CheckModeValid (SWI &3F)

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_Byte 135 (SWI &06)

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

Monitor lead I.D detection

The new architecture only detects bit 0 of the monitor lead ID. The monitor lead ID is used to determine the monitor type from the monitor lead. The ID is only used to set up an initial screen mode. During start up the exact monitor type is loaded by the ScreenModes module using a ModeInfo file. See page 96 for more information about the ScreenModes module.

ID bit 0	Description	Monitor type	Screen mode	Sync type
0	VGA capable	4	27	0
H	TV standard	0	12	1

ID bit 0	Description	Monitor type	Screen mode	Sync type
1	TV standard	0	12	1
Other	TV standard	0	12	1

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 the appropriate mode is selected.

Extensions to service call Service_ModeExtension (Service Call &50)

RISC OS 3.1 supports two formats (types 0 and 1) of the VIDC list (the first word in the list is the format type). Both the types include values which correspond directly to VIDC1 register formats. A new format (type 3) is used in RISC OS 3.X, this is independent of the video controller used. For more information about using the new format type with Service_Mode extension turn to page 119.

Extensions to service call Service_ModeTranslation (Service call &51)

This service call has been extended to allow the substitute mode passed back in R2 to be an arbitrary mode specifier. in earlier operating systems is used to pass in a mode number which was unavailable on the current monitortype, and pass back a substitute mode number.

However, the input mode will only ever be a mode number, as a mode change controlled by a pointer to a mode specifier never uses a substitute mode.

New service call Service_EnumerateScreenModes (Service call &8D)

Allows Display manager applications to find out what resolutions are available at what pixel depths. This service call is described on page 123.

Compatibility issues

There are the following compatibility issues:

VDU 23, 17, 0 - 3

This option works for both 8bpp Ncolour=63 and 8 bpp Ncolour=255 modes. However it is of little use in 8 bpp Ncolour=255 modes.

VDU 22

Using VDU 22 to select a screen mode does not allow all display modes to be chosen. You should no longer use VDU 22. The call operates as it did in RISC OS 3.1.

This call is now deprecated in favor of OS_ScreenMode which allows full access to the screen modes available on the computer.

VDU 17 and VDU 18

The calls VDU 17 and VDU 18 are no longer useful since colours may now be words rather than bytes. The SWI SYS OS_SetColour should be used instead. The colour number to be used can be found by using ColourTrans_ReturnColourNumber. You should no longer use VDU 17 or VDU 18. In 8bpp or lower colour modes it continues to work as before.

VDU 19

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.

OS_Word 9

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

OS_Word 11

Use ColourTrans_ReadPalette (SWI &4075C) in preference to this call.

OS_Word 12

Use ColourTrans_WritePalette (SWI &4075D) in preference to this call.

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 copy the other four from the significant four or set them to zero. This technique still works but only allows access to sixteen of the possible 256 intensities.

Sprite plotting colour translation tables

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.

Software vectors

New PaletteV reason codes

PaletteV has been enhanced to allow the block read and write of the palette.

ColourTrans (and other users of these reason codes) tries the new reason codes first and then falls back to using reason codes 1 (read) and 2 (write). Should these also fail it uses OS_ReadPalette/VDU19 (the old behaviour of ColourTrans). PaletteV is described on page 116.

Wimp

Wimp_SetMode (SWI &400E3)

This call took a mode number in R0 on input. It now also accepts a mode specifier (a pointer to a mode selector). In the case of a pointer to a mode selector, the Window Manager module makes a copy of the whole structure to use on future mode changes, not just a copy of the pointer.

*WimpMode

This command now allows the mode to be specified either as a number, or in the form of a mode description string, which is a textual form of a mode selector. This is reflected in the Display manager application, which also allows this form.

Software vectors

PaletteV (Vector &23)

Called whenever the palette is to be read or written. New reason codes in RISC OS 3.X are reason code 7, reason code 8 and reason code 9. For information on the other reason codes refer to the RISC OS 3 *Programmer's Reference Manual* in the chapter Software vectors page no XX.

On entry

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

Read palette entries

R0 = list of logical colours (words) or 0
R1 = bit 31 to bit 24: colour type (16,17,18,24 or 25)
bit 23 to bit 00: number of colours
R2 = pointer to memory for first flash state colours
R3 = pointer to memory for second flash state colours
R4 = 7 (reason code)

All pointers should be word aligned.

Write palette entries

R0 = list of logical colours (words) or 0
R1 = bit 31 to bit 24: colour type (16,17,18,24 or 25)
bit 23 to bit 00: number of colours
R2 = list of device colours (words)
R4 = 8 (reason code)

All pointers must be word aligned.

Gamma correction tables

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

On exit

Read palette entries

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

Registers R0 to R3 are preserved.

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

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 other flash state is undefined.

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=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.

Not all PaletteV claiments 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 claiments processing some reason codes and passing on others it does not understand.

ColourTrans

ColourTrans (and other users of these new reason codes) will read and write the palette by trying the new reason codes first and then falling back to using reason codes 1 (read) and 2 (write). Should these also fail it falls back to using OS_ReadPalette/VDU19 (the old behaviour of ColourTrans).

Reason code 9

RISC OS now allows an application to supply 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.

In a similar fashion the green and blue components are looked up in their respective tables.

The contents of the tables are set up by a call to PaletteV Reason code 9.

Service calls

Service_ModeExtension (Service Call &50)

RISC OS 3 supports two formats (types 0 and 1) of the VIDC list (the first word in the list is the format type). Both the types include values which correspond directly to VIDC1 register formats. A new format (type 3) is used in RISC OS 3.X, which is independent of the video controller used.

RISC OS 3.X registers

On entry

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

On exit

R1 = 0
R2 = is preserved
R3 is a pointer to the VIDC list (type 3)
R4 is a pointer to the workspace list if a mode number was passed in, or R4 = 0 if a pointer to a mode selector was passed in

All registers preserved if not recognised

Use

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.

If a pointer to a mode selector is passed in, the module should check that bits 7..0 of the mode selector flags hold the value 1. If they hold a different value, this implies that the structure of the rest of the mode selector may be different, so the module should pass the service on, unless it recognises the new format.

For format checking purposes, bits 31..8 of the mode selector flags are ignored, as these only pass in extra information which may be of use to some modules.

The mode selector could contain –1 as the frame rate, in which case the first matching mode should be used.

RISC OS 3 supports two formats (types 0 and 1) of VIDC list (the first word of the list is the format type). These types include values that directly correspond to VIDC1 register formats. These formats are **not** supported on RISCOS 3.X; mode extend modules for RISC OS 2 and 3 **will not work**.

A new format list (type 3) is available on the new architecture. This 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 (pixels)
12	back porch
16	left border
20	display size
24	right border
28	front porch
32	vertical sync width (pixels)
36	back porch
40	top border
44	display size
48	bottom border
52	front porch
56	pixel rate (kHz)
60	sync polarities bit 0 set for Hsync inverted bit 1 set for Vsync inverted bits 2 to 31 reserved and must be zero (if composite sync configured, the syncs are not inverted)
64	video control parameters list
n	–1 (terminator)

The control parameters list does not normally contain entries for normal video operation. These are only needed for special video operation.

The video control parameters list (offset 64) contains pairs of words (control parameter index, value) terminated by a –1 word. These control additional VIDC registers and bits in registers. 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 to 255
4	LCD offset register 1	0 to 255
5	Hi-res mode	0 = disable, 1 = enable
6	RGB pedestal enables	bit0 = R, bit1 = G, bit2 = B
7	External register [7:0]	0 to 255
8	hclk select/specify	see note A
9	rclk frequency	see note B

Note A

This value, if non-zero, forces RISC OS to select hclk as its clock input, and the value itself indicates to RISC OS what frequency is expected to be coming in on the pin (in kHz).

RISC OS then sets up the internal divider to try to achieve the pixel rate specified in the main body of the list. Normally you would make this the same, so that RISC OS would select divide by 1. The frequency is also used by RISC OS to compute the FIFO load position.

If the value is zero, then RISC OS uses rclk and vclk as per normal.

Note B

This value, if non-zero, tells RISC OS that the frequency coming in on the rclk pin is not 24MHz, but is the value specified (in kHz). This value is used to determine the moduli values for the specified pixel rate and the FIFO load position. The value is also stored in VDU variable VIDCClockSpeed. This is read by the sound system to determine the value to program the Sound Frequency Register with (since the sound system is driven by rclk).

Additional parameters passed on R4 and R5 on entry

The video bandwidth available is supplied, in bytes/second, on entry to R4. The total amount of video RAM is passed in R5 on entry.

The range of computers vary in their video capabilities. For video memory the new architecture can use 1MB of DRAM, 1MB of VRAM or 2MB of VRAM. These 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.

A mode provider should only respond to this service if the mode being selected would use no more video bandwidth than the amount in R4, and no more video memory than the amount in R5.

Service_EnumerateScreenModes (Service Call &8D)

Allows 'Display manager' applications to find out what resolutions are available at what pixel depths.

On entry

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

On exit

R2 = updated enumeration index
R3 is preserved
R4 is preserved
R5 is preserved
R6 = updated block pointer
R7 = updated size of block

Use

This service call allows modes to be enumerated. It can be issued in two ways:

- To find out how many modes there are, and how much space is needed to store them all:
- To enumerate them into a buffer. This can be done as a partial enumeration so that a fixed size buffer can be used.

Examples

The following descriptions are from the point of view of the caller

Finding out how many modes there are and how much space they need

On entry

R1 = &8D (reason code)
R2 = 0 enumeration index
R3 = monitor type
R4 = memory bandwidth available
R5 = total video RAM
R6 = 0
R7 = 0

On exit

R2 = - (number of modes)
R7 = - (amount of space needed for them)
Other registers preserved

Doing a partial enumeration

On entry

R1 = &8D (reason code)
R2 = number of modes to skip
R3 = monitor type
R4 = memory bandwidth available
R5 = total video RAM
R6 = pointer to block
R7 = size of block

On exit

If block big enough
R1 = &8D
R2 = - (number of modes in block)
R6 = pointer to byte after last filled in byte
R7 = amount of unused space in block

If block too small
R1 = 0 (Service_Serviced)
R2 = - (number of modes in block)
R6 = pointer to byte after last filled in byte
R7 = amount of unused space in block

As a client of this service call your algorithm should be:

```
Repeat
        R2 = total number of entries processed so far (ie is 0 first time
round)
        R6 = pointer to block
        R7 = size of block
        Call OS_ScreenMode reason 2 (Enumerate screen modes)
        Process block contents (from block to R6 returned from OS_ScreenMode)
Until R1 returned from OS_ScreenMode was non-0
```

As a module that wishes to respond to this service call your algorithm should be:

```
For each mode that wants to be returned
        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)
```

The registers are updated as follows:

R1 = preserved (block didn't overflow) or 0 (block overflowed)

R2 = updated enumeration index (decremented by 1 for each entry considered)

R3-R5 are preserved

R6 = block pointer, updated to point after all entries stored in the block (or stays at 0)

R7 = size of remaining free area in block (or gets decremented by amount of space needed by the entries that would have been put into the block had R6 been non-0)

The module returns information on all modes that it provides that work on the specified monitor type and which require no more than the specified memory bandwidth and no more video memory than the specified amount.

R1 on exit

If the block is of sufficient size to hold all the entries to be returned by the module, then R1 is preserved. If it is not of sufficient size, as many entries as fit are stored in the block and R1 is set to 0 (the service is claimed) to indicate that more modes are available.

Module information

For each screen mode the module stores the information in the block:

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

SWI calls

OS_SetColour (SWI &61)

Sets the foreground or background graphics colours

On entry

R0 = flags:

bits 0 - 3	graphics plotting action (see below)
bit 4	set ⇒ alter background, clear ⇒ alter foreground
bit 5	set ⇒ R1 = pattern data, clear ⇒ R1 = colour number
bit 6	set ⇒ R1 = text colour, clear ⇒ R1 = graphics colour
bit 7	set ⇒ read colour, clear ⇒ set colour
bit 8 to 31	reserved – set to 0

R1 = If R0 is bit 5	colour number (if R0 bit 5 is clear) or pointer to eight words of pattern data (if R0 bit 5 is set)
If R0 is bit 6	pointer to pattern block (if R0 bit 6 is clear)

On exit

Set	all registers preserved
Read	R0 = flags R1 = colour number or pointer to pattern block

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the foreground or background graphics colours.

This call is for internal use only.

RISC OS 3.X notes

When setting the colour all the flags are used. When reading the colour only the foreground/background flag and the text colour flag is used. A pattern block must be supplied for reading as this will be filled in with the ECF if necessary. The values returned by reading the colour are ready for passing straight to OS_SetColour to set the colour back. Reading returns the following:

Graphics colour wanted:

R0 returns:

bit	meaning
0 to 3	logical operation
4	preserved (fg/bg flag)
5	1 (pattern block flag)
6	0 (text/graphics flag)
7	0 (read/write flag)
8 to 31	preserved

R1 unchanged, pattern block filled in

Text colour wanted:

R0 returns:

bit	meaning
0 to 3	logical operation
4	preserved (fg/bg flag)
5	0 (pattern block flag)
6	1 (text/graphics flag)
7	0 (read/write flag)
8 to 31	preserved

R1 colour number

RISC OS 3.1 notes

You can obtain the colour number to use from ColourTrans_ReturnColourNumber. You can supply an eight word pattern block to use giant ECFs instead of solid colours. With this call you can define a giant pattern for both the foreground and background colours, whereas the VDU drivers only allow you to set a single pattern for both.

The graphics plotting action passed in bits 0 - 3 of R0 is as follows:

Value	Action
0	Overwrite colour on screen with colour
1	OR colour on screen with colour
2	AND colour on screen with colour
3	exclusive OR colour on screen with colour

4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND colour on screen with (NOT colour)
7	OR colour on screen with (NOT colour)
8 - 15	As 0 to 7, but background colour is transparent

Related SWIs

ColourTrans_ReturnColourNumber

Related vectors

None

OS_ScreenMode 0 (SWI &65)

Selects a screen mode

On entry

R0 = 0 (select screen mode)
R1 = mode specifier

On exit

All registers preserved

Interrupts

??

Processor mode

??

Re-entrancy

??

Use

This call selects the given screen mode.

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 a pointer to a mode selector was given, but the specified mode cannot be selected for any reason, then an error is always returned. This behaviour is different from mode selection by number, which can select a substitute mode in certain cases.

If a pointer to a mode selector is given, then any mode variables which are not specified are given sensible defaults, based on the specified resolution and pixel depth. Note that the relevant information is copied away by RISC OS, so mode selector structures need not remain valid after the call has returned.

The way that RISC OS works out the mode variables for a given screen mode is different when a mode selector is used:

When given a mode number,

- if it 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 and the module which responds to this passes back a workspace list which contains a base mode (which must be known to RISC OS) and a list of changes to mode variables.
- if it is passed a pointer to a mode selector, however, the mode variables are initialised to defaults based on the information in the mode selector block, such as the x and y resolutions and the pixel depth.
- mode variables specified in the mode selector block override these defaults.

The default values for these 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	1if 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 in the mode selector case, but only if RISC OS does not know the video timings for the resolutions/pixel depth/frame rate asked for, and in this case 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, ie palette entries get modified in groups of 16. This is so that programs which expect the old behaviour work without modification in these modes.

To gain access to fully-palette-programmable 256 colour modes, the following variables should be overridden with these values:-

Variable	Value
ModeFlags	128
NColour	255

Under these conditions, all 256 palette entries become programmable, although the initial palette setting is identical.

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 before switching screen banks. If there is sufficient memory then switching banks will work without a hitch.

Related calls

Service_ModeExtension

OS_ScreenMode 1 (SWI &65)

SReturns a mode specifier for the current mode.

On entry

R0 = 1 (return mode specifier for current screen mode)

On exit

R1 = mode specifier

Interrupts

??

Processor mode

??

Re-entrancy

??

Use

This returns the mode specifier for the current screen mode.

If the screen mode was selected by a mode number then the mode number is returned from this call, otherwise a pointer to a mode selector is returned.

OS_ScreenMode 2 (SWI &65)

Enumerates the screen modes

On entry

R0 = 2 (enumerate screen modes)
R2 = enumeration index (0 to start from beginning)
R6 = pointer to block to return data into, or 0 just count entries
R7 = size of block if R6 <>0 or zero if R6=0

On exit

R1 = 0 if service claimed, otherwise R1 non-zero
R2 = updated enumeration index
R6 = updated block pointer
R7 = size of remaining free area in block

All other registers are preserved

Interrupts

??

Processor mode

??

Re-entrancy

??

Use

This call provides a front-end to Service_EnumerateScreenModes. 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.

Note: if R6=0 on entry to this SWI, then no data is returned, but R2 will have been updated to the number of entries that would have been returned, and R7 will have been decremented by the number of bytes that would have been put into the block, therefore – R7 is the size of block that will be needed to hold all the entries.

Each entry stored 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 7..1 = mode info format specifier (zero for this format) bits 31..8 = additional mode info flags (must be zero)
8	x-resolution (in pixels)
12	y-resolution (in pixels)
16	pixel depth (as per mode selector)
20	frame rate (Hz) (to the nearest integer)

In the future it is possible that different format entries may be returned by modules, therefore callers of this SWI should check that bits 7..0 of the mode provider flags contain 1 before extracting the other information in this block. If these bits contain a different value which is not recognised by the caller, then the entry should be skipped by adding on the size field at offset 0. For format checking purposes, bits 31..8 should be ignored.

Related calls

Service_EnumerateScreenModes

OS_ScreenMode 3 (SWI &65)

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

*Commands

*LoadModeFile

Loads a ModeInfo file into memory

Syntax

*LoadModeFile *filename*

Parameters

filename

Use

This command loads a ModeInfo file into memory. If the file contains valid information, it sets the current monitortype 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.

7 Monitor power saving

5

Introduction

Government agencies and independent organisations worldwide are involved in setting limits or goals for power consumption in office equipment. Desktop computers are one of the primary targets for this effort. The display consumes a significant portion of the power used in a desktop computer system. These agencies would like to decrease energy use from computer displays specifically, in order to slow the growth in overall demand for electric power.

Display Power Management Signalling (DPMS) is a proposed standard produced by VESA (Video Electronics Standards Association), which provides a common definition and methodology by which a display controller can send a signal to the display that enables it to enter various power management states.

The states are distinguished by the presence or absence of pulses on the horizontal and vertical sync lines.

State	Power saving	Recover time	Horiz sync	Verti 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.

DPMS is likely to be adopted by most major monitor manufacturers, and the new RISC OS can take advantage of this mechanism where supported by the monitor.

Software changes to support DPMS

RISC OS 3.1 provides support for blanking the screen after the computer has been left untouched for a certain amount of time, in order to avoid phosphor dot burn-out. RISC OS 3.x incorporates DPMS into this, so that when the screen blanks, it can select one of the reduced-power states above. The current

screen-blanking mechanism blanks the video, but leaves the sync pulses active. This option needs to be preserved, for the case of monitors which do not support DPMS and which would get upset if syncs were removed.

Also, since it is not defined by the DPMS proposed specification which of the three reduced-power options a DPMS monitor must support, it is probably necessary to be able to select any of them.

This gives the following four options to control:-

1 DPMS disabled - screen blank just blanks video

2 Screen blank enters 'Stand-by' mode

3 Screen blank enters 'Suspend' mode

4 Screen blank enters 'Off' mode

(The overall enabling/disabling of screen blanking altogether is controlled separately.)

How are these options controlled?

The information controlling the DPMS state when blanked is held in the monitor description file on the hard disc.

Changes required for this are as follows:- The format of a video control parameters list is expanded to include a new control index (11), whose value is in the range 0 to 3, with the following meanings:

0 DPMS disabled - screen blank just blanks video (default if this control index is absent)

1 Screen blank enters 'Stand-by' mode

2 Screen blank enters 'Suspend' mode

3 Screen blank enters 'Off' mode

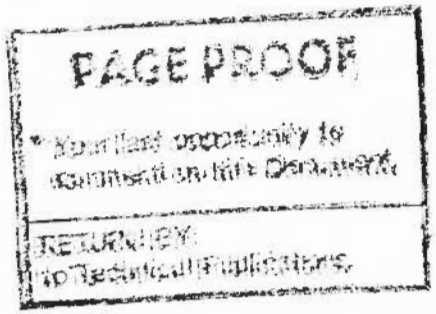
The ScreenModes module is changed to allow an optional line after the 'monitor_title' line which reads:

DPMS_state: n

where n is a number in the range 0 to 3, with the same meanings as above. This line (which is global to the file) causes the ScreenModes module to append a video control parameters list to all VIDC lists it passes to the kernel, with the above-mentioned control index and the specified value.

Any monitor description files on the hard disc which are designed to drive monitors supporting DPMS (such as the Acorn AKF60) are modified to include the extra line above.

One disadvantage with this scheme is that for monitors that support all 4 power-reduction states, it's difficult for the user to try-out selecting different states, since he has to edit the monitor description file (and he has to know where this information is). Also, if a user has a monitor which doesn't have a dedicated file on the disc, he may not know that he can enable the DPMS feature.



8 The colour picker

Introduction

The new architecture uses the improved video chip, the VIDC20. This chip and the supporting architecture allow screen displays of 8bpp, 16bpp and 32bpp. These colour depths can be translated into numbers of colours:

8bpp	256 colours
16bpp	32 thousand colours
32bpp	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. It is currently used by (future versions of) !Draw, !Paint and several third party applications.

This chapter describes how the client application and the colour picker module interact.

Terminology

Colour Picker: The Colour Picker module provides a colour selection dialogue box for applications to use. This is not a Wimp task. The colour picker makes use of the filter mechanism to receive events for its windows.

Client: The application which invokes the Colour Picker. The client is a Wimp task.

Colour descriptor: The structure giving the full details of a colour. It includes a base RGB value, a colour model and the parameters for the chosen colour in terms of the colour model.

Overview

Here is a summary of how the colour picker operates:

- The client application communicates with the colour picker by calling SWIs
- Whenever a new colour selection dialogue is opened, the colour picker module installs a Wimp pre-filter and post-filter box around the client application.
- The colour picker module maintains the colour dialogue box by intercepting Wimp events directed to that window.
- The colour picker module passes user colour choices and other information to the client using Wimp messages.

The colour descriptor

In the simplest form colours are passed as simple 1 word RGB value. An application that makes full use of the facilities can store the full colour descriptor. The colour descriptor includes the basic RGB value, as well as a structure giving colour model, and the colour value as represented in that colour model. Other information can also be defined.

SWIs

SWIs are used for application to Colour Picker communications; messages for communication the other way. Wimp events are received by the Colour Picker becoming a filter.

User interface

The user interface for three colour models, (RGB, HSV and CMYK) are described in the chapter entitled *User interfaces* on page 293.

Technical details

Colour picker application program interface

Communication between the application and the colour picker is performed by the application calling the colour picker and the colour picker responding with SWI calls or messages.

The colour picker uses the following SWIs

- SWI ColourPicker_OpenDialogue
- SWI ColourPicker_CloseDialogue
- SWI ColourPicker_UpdateDialogue
- SWI ColourPicker_ReadDialogue

These SWIs are used by the application to control a colour selection dialogue. A colour selection dialogue is identified by its window handle. The SWIs are documented starting on page 150.

A typical colour selection is seen by the application as follows:

```
<App receives colour selection request by user>
<App prepares structure describing dialogue>
 _swi(ColourPicker_OpenDialogue,
        _IN(0)|_IN(1)|_IN(2)|_IN(3)|_OUT(0), /* regs in/out */
        0, my_task, 0, &mydialogue, /* values in */
        &mydialoguehandle): /* values out */
<User makes colour selection>
<App receives ColourPicker_Action>
<App applies colour information>
```

Colour picker API Messages

These are generated by the Colour picker in response to Wimp events on the colour dialogue.

Message_ColourPickerChoice (message &46141)

Issued to the application when the user makes a definite choice of colour, by clicking Select or Adjust on the OK button of the dialogue box.

offset	
20	handle of the dialogue box concerned
24	flags: bit 0: 'none' chosen
28	colour block chosen

This message is always sent if the relevant event happens (though if the user closes the dialogue box by pressing Cancel, it will not be sent at all). When flags bit 0 is set, signifying that 'transparent' was chosen, the colour descriptor *will* be present. The colour descriptor gives the state of the dialogue so that a sensible default may be given next time the dialogue is used.

Message_ColourPickerColourChanged (message &46141)

Receipt of this message indicates that the user has selected a new colour. Its frequency of transmission is set by the SWI ColourPicker_OpenDialogue.

offset

20	handle of dialogue box concerned
24	flags
	bit 0 'none' chosen
	bit 1 drag in progress
28	colour block chosen

Issued to the application when the colour displayed in the dialogue box changes, in accordance with the setting of the 'button type' of the dialogue.

Message_ColourPickerCloseDialogueRequest (message &46142)

offset

20	handle of the dialogue box concerned

Issued to the application when the user dismisses the dialogue box, by clicking Select on the OK or Cancel icons. The application should respond by calling ColourPicker_CloseDialogue with the given handle.

If the dialogue box was opened with the 'transient' flag of ColourPicker_OpenDialogue set, then it will not see this event for that box, as the WIMP will close the window automatically.

Colour descriptor

This structure is used to pass colour choice information between the client and the Colour Picker.

byte	meaning
0 - 3	&bbggrr00
4 - 7	number of bytes of data following this word
(The remainder of this structure is optional)	
8 - 11	colour model number
12..	colour information for the given colour model (see below)

When the client passes this to the colour picker, the client may omit all data from byte 8 onwards. In that case the block would look like this:

byte	value
0 - 3	word, &bbggrr00
4 - 7	0

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 information for specific colour models

Colour Model 0: RGB

Word 0: R
Word 1: G
Word 2: B

Data length = 12
Value range &00000000 to &00ffffff. This gives 256 equally sized intervals.

Colour Model 1: CMYK

Word 0: C
Word 1: M
Word 2: Y
Word 3: K

Data length = 16
Value range &00000000 to &00ffffff. This gives 256 equally sized intervals.

Colour Model 2: HSV

Word 0: H
Word 1: S
Word 2: V

Data length = 12
Value range for H is &00000000 to &0167ffff. This gives 360 equally sized intervals.
Value range for S and V is &00000000 to &00010000.

All colour quantities in the above are in ColourTrans style fixed point 16+16 numbers.

SWI calls

ColourPicker_RegisterModel (SWI &46140)

Called by a colour selection model in its initialisation code

On entry

R0 = model number
R1 Pointer to the model data
R2 Pointer to the model workspace

On exit

All registers preserved

Interrupts

?

Processor Mode

?

Re-entrancy

?

Use

This SWI must be called by a colour model module in it initialisation code. In conjunction with the service call Service_ColourPickerLoaded, this ensures that the Colour Picker always knows about all the colour models, no matter which order they are started in.

Related SWIs

?

Related vectors

?

ColourPicker_DeregisterModel (SWI &46141)

Called by a colour selection model in its termination code

On entry

R0 = model number

On exit

All registers preserved

Interrupts

?

Processor Mode

?

Re-entrancy

?

Use

This SWI must be called by a colour model module in it termination code. It informs the Colour Picker that the colour model is no longer available.

Related SWIs

?

Related vectors

?

ColourPicker_OpenDialogue (SWI &46142)

Opens and creates a colour picker dialogue box for the client

On entry

R0 = flags
bit 0: dialogue box is t be transient
All other bits reserved (must be set to 0)

R1 Pointer to a block

Offset	
0	flags bit 0: dialogue box is to offer a 'None' button bit 1: dialogue box is to have the 'None' button selected bits 2, 3: dialogue box 'button type,' defining when Message_ColourPickerColourChnaged is issued for it: 0: Never issued 1: On any change, except during drags. Drags give message at drag end. 2: On any change, including during drags
4	pointer to the title to be used, or 0 for a default title
8	x0
12	y0
16	x1
20	y1 of the bounding box of the visoble area of the dialogue box; only (x0, y1) are honoured. (x1, y0) should be set to large positive and negative values respectively
24	xscroll
28	yscroll of the dialogue box. Only used if the visible area provided in offsets 8 to 20 is too small (which should never be the case); in which case the dialoge box gets scroll bars automatically
32	a colour block defined as follows:

Offset	data
0	0
1	blue value (0, ..., &FF)
2	green value
3	red value
4	size of remainder of this block (optional)
8	colour model number (as in the service call)
12	other model dependent data

On exit

R0 = dialogue box handle

Interrupts

?

Processor Mode

?

Re-entrancy

?

Use

This SWI is used by an application that wants to display a Colour Picker dialogue box to the user, for interactive choosing of a colour. The handle is used as an argument to the other ColourPicker SWI's, and also in messages that the ColourPicker module sends to the application to provide feedback on the user's selection of a colour. The colour descriptor specifies the initial settings placed into it. The dialogue will always have an OK and Cancel button. If offset 0 bit 0 bit is set, then a 'None' button appears between the colour patch and the Cancel button. The button can be automatically selected if offset 0 bit 1 is set.

Related SWIs

?

Related vectors

?

ColourPicker_CloseDialogue (SWI &46143)

Called by the application to close the dialogue.

On entry

R0 = flags: all bit are reserved and must be set to 0
R1 = dialogue box handle

On exit

?

Interrupts

?

Processor Mode

?

Re-entrancy

?

Use

Abandon a dialogue which is currently in progress. This may also be done by the Wimp if the dialogue is a transient one. It is normally called in response to Message_ColourPickerCloseDialogueRequest.

Related SWIs

?

Related vectors

?

ColourPicker_UpdateDialogue (SWI &46144)

Allows the client to change the colour dialogue's state.

On entry

R0 = flags

bit	meaning
0	whether the dialogue box offers a 'None' button
1	whether 'None' button is currently selected
2	the button type of the dialogue box
3	the visible area of the dialogue box (only (x0, y1) are honoured)
4	the scroll offsets (not honoured)
5	the window title
6	the colour (r,g,b triplet only)
7	the colour model (including all the optional data that the colour model requires)

R1 = dialogue handle
R2 is a pointer a block as in ColourPicker_OpenDialogue

On exit

?

Interrupts

?

Processor Mode

?

Re-entrancy

?

Use

This SWI changes some or all of the contents of the dialogue box whose handle is given. Only the parts of the box indicated by the flags word are updated. If bit 7 is set, bit 6 is ignored: the (r, g, b) triplet is calculated from the data in the colour model block. The transparent button may be added or removed and its setting adjusted. The title, colour model and setting may also be adjusted independently

of each other. If the colour setting is updated, but not the colour model and the current colour model isn't that in the colour descriptor then the colour descriptor's RGB word will be used and the colour model left alone.

Related SWIs

?

Related vectors

?

ColourPicker_ReadDialogue (SWI &46145)

Allows the current state of the dialogue to be read without closing or altering the window.

On entry

R0 = window handle of dialogue box
R1 is a pointer to the buffer, or 0 (to read size)

On exit

R1 = size required, or preserved if non- zero

Interrupts

?

Processor Mode

?

Re-entrancy

?

Use

Fills the given buffer with a block as in ColourPicker_OpenDialogue. The block is assumed to be big enough: to find out what size is required, R1 = 0 should be specified. This means you must call the SWI twice in order to use it safely. This size may change when the colour model changes, so care should be taken to always ask for the block size before getting the colour.

Related SWIs

?

Related vectors

9 Keyboard and mouse

Introduction

One of the main changes in the new architecture is the removal of the Acorn keyboard interface from the kernel and its replacement with a standard PC AT-style keyboard driver provided as a separate module.

The standard quadrature mouse driver has also been removed form the kernel and is now a module. Additionally there is a serial mouse driver module that is used if you connect a standard PC-type (Microsoft or Mouse Systems) mouse to the serial port.

Multiple devices

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.

Overview

Keyboard interface

The keyboard uses the interfaces in the IOMD chip; these are similar to the interfaces in the previous generation IOC chip.

The keyboard module drives the keyboard. The kernel is informed when a key is pressed or released; the kernel also tells the keyboard driver the state of the keyboard LEDs.

Reset combinations

The power-on and reset combinations for RISC OS 3.X have been changed to rationalise a previously confusing set of options. The following scheme is now used.

Key combination	Function
Power-on	Normal reset, use boot options
Ctrl-break	Normal reset, use boot options
Reset	Normal reset, use boot options (use when keyboard does not respond)

The following modifiers can be used in conjunction with the above resets:

Shift	Reverse action of configured boot option
* (on keypad)	Use boot options but boot to command line (instead of the configured language)

The following modifiers can be used during a power-on reset only:

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)	Configure monitor type
. (on keypad)	Configure auto monitor type, sync and mode

All resets are now effectively hard resets. The previous concept of hard and soft resets is no longer used. For backward compatibility, pressing Shift-Break causes the same action as Shift-Ctrl-Break.

Quadrature mouse interface

The mouse interface no longer resides in the kernel. It is now incorporated into a module. The interface can use different types of pointing devices, including a serial mouse. Multiple pointing devices can exist on the computer, but only one can be active at any one time.

IOMD

IOMD does not provide interrupt support for mouse input. Instead IOMD 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 where these registers are polled regularly by the mouse driving software.

Serial mouse interface

The module SerialMouse drives a serial mouse connected to the serial port of the computer. The module uses the same interfaces as the quadrature mouse driver see page xxxx.

The serial mouse driver communicates with serial mice which transmit data in one of two formats. These are the Microsoft compatible format and the Mouse Systems Corporation compatible format.

The serial mouse driver only accepts data from mice communicating in a stream mode operating at 1200 baud. Some devices support higher rates, but these higher rates are not supported.

Technical details

Keyboard Interface

The RISC OS 3.1 keyboard code has been removed from the kernel.

The kernel is now informed of key presses and releases by the keyboard driver. The kernel informs the keyboard driver of LED status.

The function of the keyboard driver is to drive the keyboard device, all other functionality is dealt with by the kernel.

The IOMD chip provides an interface that is similar to the IOC chip used in the previous range of computers. This 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.

Keyboard module

The keyboard driving software now resides in a separate module. The module communicates with the kernel telling it which keys are pressed and released. Additionally there is a call to set the keyboard LEDs and a call to initialise the keyboard.

The PC keyboard driver claims the keyboard device interrupts using SWI OS_ClaimDeviceVector. This then uses the software vector Keyv to notify the kernel of keys pressed or released. This vector has not been used in RISC OS and so it has now been redefined.

Vector KeyV (Vector &13)

KeyV is used to communicate between the kernel and the keyboard driver. For a detailed description of the vector turn to page xx

Keyboard driver

The keyboard is the most used part of character input and its driver the most complex. In principle it is simple enough but many features are changeable and key presses can be looked at in a number of ways. There are three parts to the keyboard driver which are described below.

Keyboard Device Driver

The keyboard device driver handles the keyboard interrupt and low-level control. It is concerned only with driving the specific input device being used. The supplied device driver is for a PC-AT keyboard but can be replaced by a custom version if required (eg. for a special needs input device).

Keyboard Handler

The keyboard handler 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).

System Keyboard Driver

This part resides in the RISC OS kernel and it binds the keyboard device driver and keyboard handler together. The keyboard device driver passes low-level key numbers to the system which calls on the keyboard handler to convert them into a recognisable form. The system also debounces key presses, keeps track of keys down and generates auto-repeats of keys at the configured rate.

Basic operation

At a basic level, the keyboard works like this:

1. One or more keys are pressed, which cause an interrupt.
2. The keyboard device driver receives a code from the keyboard and converts it into a low-level key number (as used in key up/down events).
3. The low-level key number is passed to the kernel via vector KeyV.

The kernel calls on the keyboard handler to convert the low-level key number to a more recognisable form. This can be:

- an ASCII character.
- a non-ASCII character such as a function key or arrow.
- a special key such as Escape or Break

Quadrature mouse driver

The RISC OS kernel is responsible for:

- Registering the mouse buffer with the buffer manager
- All mouse bounding
- Updating the mouse pointer on the display and responding to OS_Mouse calls.

PointerV (Vector &26)

A new vector PointerV is used to communicate between the kernel and the pointer device driver.

The Mouse module drives a quadrature mouse using the IOMD interface. This module claims PointerV on initialisation using SWI OS_Claim, passing the address of a sub-routine which conforms to the interface. This sub-routine polls the mouse position registers in IOMD and returns mouse movements, calculated by subtracting the previous values of these registers from the new ones.

The kernel calls the registered sub-routine on VSync and scales the mouse movements depending on the configured mouse step. The pointer position is updated on the display.

The active pointer device driver calls KeyV to notify the kernel of buttons being pressed or released. The kernel treats these as any other key including debouncing.

SWI OS_Pointer (SWI &64)

So that the pointer device type can be selected during machine operation a new SWI OS_Pointer is used.

This SWI is used to obtain the type of the pointer device currently in use or to select the pointer device type to be used. The selected device type is passed to claimants of PointerV when the system calls the vector to update its record of the pointer position. Selecting a new type causes PointerV to be called with reason code 2 (pointer type selected) so that drivers can enable or disable.

Configuring the mouse type

The command *Configure MouseType selects the type of pointer device, quadrature or serial. The configured device type is stored in CMOS RAM and is used by the kernel in PointerV calls after a reset or power-on.

Serial mouse driver

On initialisation this module claims PointerV and responds to requests for device type 1 or 2 (serial mouse).

When the serial mouse driver receives a PointerV call with reason code 2 (pointer type selected) and the device type is 1 or 2 the driver configures the serial device using OS_SerialOp and opens device 'serial:' for input.

The driver also claims TickerV and processes any data received by the serial device on centiscecond clock ticks. Mouse movements and button states are amalgamated until PointerV is called with reason code 0 (request pointer device state) at which time a report will be returned to the kernel.

The serial mouse driver does not prevent the reconfiguration of the serial port while the driver is active; however it ensures that the 'serial:' device is reopened if it is closed by an external source.

If the serial mouse driver receives a PointerV call with reason code 2 (pointer type selected) and the device type is not set to 1 or 2, the driver will release vector TickerV and close device 'serial:'.

The code which is called on TickerV re-enables interrupts so that interrupt latency is not adversely affected. In order to prevent reentrancy, the code sets a flag while it is being executed.

Data formats

The serial mouse driver communicates with serial mice which transmit data in one of two formats.

Microsoft

The first class of mice are Microsoft compatible. They send a data report 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 = Key data, Left, right, middle, 1 = key down
X7 – X0 = X distance 8 bit value, –128 to +127
Y7 – Y0 = Y distance 8 bit value, –128 to +127
DT4 to DT0 = device type (0 = mouse, all others reserved)

Not all three button mice generate the 4th byte in the report; in some cases when the middle button is pressed L and R are both set to 1. With mice that generate a 4th byte for the middle key, L and R are not affected. The driver detects the state of the middle key in both cases.

Y movement is negative to the south and positive to the north.

Mouse Systems

The second class of mice are Mouse Systems Corporation compatible and sends reports in Five Byte Packed Binary 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 = Key data, Left, right, middle, 0 = key down
X7 – X0 = X distance 8 bit value, –128 to +127
Y7 – Y0 = Y distance 8 bit value, –128 to +127
DT4 to DT0 = device type (0 = mouse, all others reserved)

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 and cannot be discarded.

Y movement is negative to the south and positive to the north.

Protocols

The serial mouse driver only accepts data from mice communicating in a stream mode operating at 1200 baud. Some devices support higher rates but these must be selected by sending a command to the mouse and are not supported. The mouse generates reports at rates from 10 reports/sec to continuously, with no intervals between successive reports. A change in the status of the buttons generates a report immediately. When the mouse is motionless and the button status is unchanged no reports are sent.

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.

Software vectors

KeyV (Vector &13)

Used to communicate between the kernel and the keyboard driver.

On entry

R0 = reason code (see below)
R1 = reason code dependent (see below)

On exit

All registers preserved

Interrupts

Interrupt status remains unchanged
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Use

When a keyboard driver initialises successfully, it calls this vector with reason code 0 to notify the system that the keyboard is present.

The currently defined keyboard i.d. values are:

- 1 Archimedes keyboard
- 2 PC-AT keyboard

Reason code 1 and 2

The keyboard device driver calls this vector with reason code 1 or 2 to notify the system of keys released or pressed. The key code is the same as those given in key up and key down events.

Reason code 3 and 4

A keyboard device driver should also claim this vector. The kernel will call this vector with reason code 3 to notify the driver of LED state. The kernel will call this vector with reason code 4 to enable keyboard device drivers.

A keyboard device driver does not send any keys using reason codes 1 or 2 until it has received this call. Any calls made before the kernel has issued the enable call will be ignored.

The table below shows the reason codes that are passed.

R0	R1
0	keyboard id (keyboard present)
1	key number (key released)
2	key number (key pressed)
3	status flags (0 = LED off, 1 = LED on)

bit	meaning
0	scroll lock
1	num lock
2	caps lock
3	LED states
4	keyboard enable
5-31	reserved (set to 0)

The keyboard driver assumes that the keyboard sends scan codes from IBM-MF compatible code set 2 (standard AT code set). If the keyboard does not support this set then pressing certain keys may produce unexpected results.

The PC keyboard scan codes are converted to low-level key numbers before the vector is called.

To support the current scheme for configuring auto-repeat delay and repeat rate, the auto-repeat capability of PC keyboards is not used. Keys are repeated by the kernel using the existing mechanism. The PC keyboard driver keeps a table of flags for key states. If a key is already flagged as being down, KeyV is not called.

If you wish to use some other device with this vector, contact Acorn Technical support for a keyboard number allocation.

Related SWIs

OS_ClaimDeviceVector

PointerV
R0 = 0
(Vector &26)

Used to communicate between the kernel and the pointer device driver to request the pointer device status.

On entry

R0 = reason code 0 (request pointer device status)
R1 = device type

On exit

R2 = signed 32-bit X movement
R3 = signed 32-bit Y movement

All other registers preserved

Use

PointerV is called by the kernel with reason code 0 on Vsync to obtain the latest pointer device movements. The returned values are used to update the pointer position.

The signed 32-bit X and Y values are the amounts by which the pointing device has moved since the last time PointerV was called.

The device type passed in R1 determines which claimant of PointerV responds to the request.

Current device types are:

- 0 Quadrature mouse.
- 1 Microsoft compatible mouse.
- 2 Mouse Systems Corporation compatible mouse.

If a claimant of PointerV is called but does not understand the device type then the call should be passed on. If the claimant responds with a report then the call is intercepted.

When a pointer device driver initialises it checks the configured device type using SWI OS_Pointer and if it is one that is understood by the driver then the device should be enabled.

The system treats mouse buttons as if they were keys on the keyboad. a mouse driver should notify the system of button presses by calling KeyV with one of the following key codes:

Left	(Select)	&70
Centre	(Menu)	&71
Right	(Adjust)	&72

The kernel will debounce these keys as it does for all others.

If you wish to use some other device with this vector, contact Acorn Technical Support for a device type number allocation.

PointerV
R0 = 1
(Vector &26)

Used to communicate between the kernel and the pointer device driver to identify pointer types.

On entry

R0 = reason code 1 (identify pointer types)
R1 = pointer to device type record, 0 for first recipient

On exit

R1 = pointer to driver's device type record list

All other registers preserved

Use

When a pointer device which claims PointerV receives this call, it creates a linked list of device type records, one for each type it supports:

Next pointer	4 bytes
Flags	4 bytes bits 0 to 31 reserved (set to 0)
Device type	1 byte
Text	null terminated menu text

It stores the R1 value passed to it in the 'next pointer' field of the record at the tail of the list and set R1 to point to the head of the list before passing on the call. This call must not be intercepted. The records must be claimed from the RMA and will be freed by the caller. The text should be no more than 30 characters.

PointerV
R0 = 2
(Vector &26)

Used to communicate between the kernel and the pointer device driver to give the pointer type selected.

On entry

R0 = reason code 2 (pointer type selected)
R1 = device type

On exit

All registers preserved

Use

When a device type is selected by SWI OS_Pointer (see page xx), this call is issued. All pointer device drivers which do not understand the device type should disable and the driver which understands the device type should enable.

This call must not be intercepted.

SWI calls

OS_Pointer
(SWI &64)

Allows the pointer device to be selected during machine operation

On entry

R0 = reason code 0 (get pointer type)

or

R0 = reason code 1 (set pointer type)
R1 = pointer device type

On exit

R0 = pointer device type (reason code 0)

or

All registers preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI can be used to obtain the type of the pointer device currently in use or to select the pointer device type to be used. The selected device type is passed to claimants of PointerV in R1 when the system calls the vector to update its record of the pointer position. Selecting a new type causes PointerV to be called with reason code 2 (pointer type selected) so that drivers can enable or disable.

Related SWIs

None

Related vectors

PointerV

* Commands

*Configure MouseType

Selects the type of pointer device, quadrature or serial device.

Syntax

```
*Configure MouseType <type>
```

Parameters

0 to 2

Use

Selects the type of pointer device, quadrature or serial device. !Configure now calls PointerV with the reason code 1 (identify pointer types) when the mouse configuration window is opened. This call returns a pointer to a linked list of device type records containing device numbers and a text description. This text is used to construct a menu of device types. The new device type is selected by calling SWI OS_Pointer.

The configured device type is stored in CMOS RAM and is used by the kernel in PointerV calls after a reset or power-on.

Example

```
*Configure MouseType 0
```

Related commands

*Configure
OS_Pointer (SWI &64)

10 Expansion card support

5

Introduction and Overview

The new architecture expansion card interface has been enhanced in several ways, it now supports:

- 32 bit wide data paths
- 16MB address space
- A dedicated Network card interface
- Direct Memory Addressing (DMA)

This chapter only covers the changes that have been made in order to support these enhancements. For a description of the whole of the RISC OS 3 expansion card interface you should also read the main RISC OS 3 *Programmer's Reference Manual*.

This chapter does not tell you how to design expansion card hardware. Full hardware details are given in the Technical Reference Manual; this is available from your dealer.

Technical details

The design of the new architecture expansion card interface has added two new areas:

- The extension of the existing Expansion card bus includes a directly mapped area of 16 megabytes.
- The introduction of a new format card and interface for network expansion.

The Expansion card bus is electrically capable of having ROMs (or EPROMs) connected. These ROMs can then be read by the PoduleManager for the operating system.

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 given by the memory manager. The logical and physical addresses and the area's size are passed to its clients. These addresses are stable as long as the machine configuration is stable. Software clients of the PoduleManager read these addresses once after reset.

ROMs in EASI space

Having a ROM in EASI space removes the need for a loader and for the paging register. Access to the entire ROM address space is faster. The ROMs are only 8 bits wide and are copied once at start up into RAM; not having loaders frees-up ROM space.

The format for ROMs in EASI space is the same as that for ROMs in the normal expansion card space. However since the size restriction is lifted there is no need to have a second Chunk directory accessed through the loader. The standard header information must be present including;

- Identification byte
- Manufacturer's identification number
- Product identification number
- Width field
- Interrupt relocations
- Chunk directory
- Description chunk.

Although the ROM is in the EASI space the interrupt relocations are still relative to the base of expansion card space. The chunks in an expansion card space ROM can be enumerated before an EASI space ROM.

ROMS on the network card

The network card interface specifies how a ROM may be attached to the card. It also describes the paging scheme.

The format for a ROM on the network card is the same as that for ROMs in the normal expansion card space. However since the loader is effectively loaded before the enumeration begins there is no need to have a second Chunk directory. The standard header information must be present including:

- Identification byte
- Manufacturer's identification number
- Product identification number
- Chunk directory
- Description chunk.

The interrupt relocations must be present and be all zeros.

Simple expansion card identity

Some network products are allocated simple expansion card identification values for use in the Expansion Card Identity low byte (ECId). The assignmnts are listed on page 4-120 of the RISC OS 3 *Programmer's Reference Manual*.

When ID<0..3> are all zero the normal extended ID fields are present however values 1 to 15 may be allocated to specific products. Once allocated specific software drivers may assume specific hardware when they read the ECId assignments, using SWI Podule_ReadID.

Support is also included for providing a description for allocated non-extended simple IDs. This extends the implementation of *Podules.

It is also available from the new SWI Podule_ReadInfo. The format of the token is the string 'Simple' followed by a single hexadecimal digit from 1 to F. For example:

Simple1:Acorn Econet.

The network card

The network card is the highest numbered expansion card. It is the fifth expansion card and has the number 4; it is last in the printout from *Podules.

This number (4) can be used with all SWIs except Podule_CallLoader and Podule_WriteBytes. This is because the loader isn't valid and the ROM space is treated as read only.

The SWI Podule_ReadBytes reads the ROM image via the built in loader. The SWIs Podule_RawRead and Podule_RawWrite access the device address space. It is the device address that is returned by SWIs like Podule_HardwareAddress and Podule_HardwareAddresses.

Extending the ROM sections

Currently expansion card SWIs (with the single exception of SWI Podule_ReturnNumber) use register 3 to indicate which expansion card (or extension ROM) to access. Currently this is referred to as the ROM section and has one of the following values/meanings:

ROM section	Meaning
-1	System ROM
0	Expansion card 0
1	Expansion card 1
2	Expansion card 2
3	Expansion card 3
4	**Network card**
-2	Extension ROM 1
-3	Extension ROM 2
-4	Extension ROM 3

This is extended to understand ROM section 4 as the network card.

As it is currently implemented the manager takes a liberal attitude to the value passed in R3, if it is a hardware base address, (as returned by Podule_HardwareAddress or Podule_HardwareAddress) whether combined with a CMOS address or not, this is also acceptable.

The 'formal definition' of what is acceptable in register 3 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()
WHEN Value = EASIPhysicalBase(0..7): Expansion Card()
OTHERWISE Error "Bad Expansion Card or Extension ROM number"
ENDCASE
```

SWI calls

SWI Podule_ReadInfo (SWI &4028D)

This call returns data specific to the expansion card it was requested for.

On entry

R0 = bitset of required results (see below)
R1 pointer to buffer to receive word aligned word results
R2 length in bytes of buffer
R3 any recognisable part of expansion card addressing;
e.g. expansion card number, Base address, CMOS address, New base address

On exit

R0 Preserved
R1 Preserved, a pointer to results in order (lowest bit number at the lowest address)
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 expansion card it was requested for. The returned data is in single words, which are placed into the user supplied buffer at word intervals. 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.

Bitset in R0, values

Bit 0 ==> Expansion card/Extension ROM number
Bit 1 ==> Normal (synchronous) base address of hardware
Bit 2 ==> CMOS address
Bit 3 ==> CMOS size in bytes
Bit 4 ==> Extension ROM or network ROM base address
Bit 5 ==> Combined hardware address
Bit 6 ==> Expansion card ID
Bit 7 ==> Expansion card product type
Bit 8 ==> Pointer to description (zero for no description)
Bit 9 ==> Pointer to description (pointer to "" for no description)
Bit 10 ==> Logical address of EASI space
Bit 11 ==> Physical address of EASI space
Bit 12 ==> Size of the EASI space in bytes
Bit 13 ==> Logical number of the primary DMA channel
Bit 14 ==> Logical number of the secondary DMA channel
Bits 15 to 31 are reserved and must be zero.

The setting of bits 15 to 31 is not permitted; if they are set then an error condition is returned.

This SWI is intended to supersede other Expansion card SWIs such as Podule_HardwareAddress.

Related SWIs

Podule_ReadID, Podule_ReadHeader, Podule_HardwareAddress, Podule_HardwareAddresses, Podule_ReturnNumber.

Related vectors

None.

SWI Podule_SetSpeed (SWI &4028E)

This call is used by client code that need faster (type C) access to its hardware.

On entry

R0 speed required:
- 0 ⇒ No change
- 1 ⇒ IOMD timing type A
- 2 ⇒ IOMD timing type C

R3 Any recognisable part of expansion card addressing:
- e.g. expansion card number, Base address, CMOS address, New base address

On exit

R0 Current speed setting
R1 Preserved
R2 Preserved
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 is used by client code that needs to have faster (type C) access to its hardware. The kernel initialises all expansion cards' access speed to type A.

Related SWIs

None.

11 AUN Acorn Universal Network

5

AUN overview

The AUN software 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 interfaces – both to users and to programs – so your users won't need to learn new skills, and your existing network programs should continue to work. AUN will work over your existing Econet network. However, it also gives you access to the wide range of networking technologies that support TCP/IP, and consequently a choice of prices and performance. For example, it gives you access to Ethernet, which offers a far higher performance than Econet, and which also – like TCP/IP – offers the benefits of being an industry standard.

Consequently, any migration from Econet to a faster technology such as Ethernet can be done on a step-by step basis, as required and as budgets allow, with minimum disruption. You can replace parts of your Econet with Ethernet, or add new segments of Ethernet; the only apparent change will be in the network's speed. Your investment in existing equipment and training will be maintained.

AUN enables you to transparently link together different networks – such as Econet, Ethernet, and third party networks – to build up a truly site-wide network system. However, AUN can also meet more modest requirements; a network may consist of as little as a single segment of Ethernet cable joining two or three machines on a benchtop, with a single machine doubling as a file and print server.

Furthermore, AUN's use of the TCP/IP standard supports the concept of Open Systems. Your 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.

Using an AUN network

As stated above, a key feature of the AUN software is that **its user interface is the same as that used by existing Econet software**. The only change to software that you'll notice is that the network is now referred to as a 'Net' rather than as an 'Econet'. Your users can continue to refer to file servers and to print servers by the same names that they've always used.

Your investment in your users' skills and training is not lost. As you adopt improved networking technology that offers higher speeds and capacity, your users can immediately benefit from and use it. All they'll notice is that things have got faster!

The only restrictions are that:

- **Older types of file and print servers cannot be accessed from Ethernet.** For full details see the section entitled *Coexistence with existing machines* on page 187.
- Your users may have problems with certain types of file and print servers if they refer to them by their two byte *net.station* numbers rather than by their names. **You should ensure your users refer to servers by name.**

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*.

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 software for your site.

You'll find it easiest if each network's name identifies its location within your site. For example, you may choose to use the name of the department that each network services:


```
compsciA
compsciB
science
art
business
```

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.
- 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.

It is important that you grasp the distinction between a net and a network; this guide 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 you 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 you 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), you 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 – you 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.

A gateway station, like a file server or a print server, is an important part of your site's network infrastructure. All have important configuration files that must be kept secure from users; they should not normally be used as personal workstations. You can combine some or all of these functions in a single station.

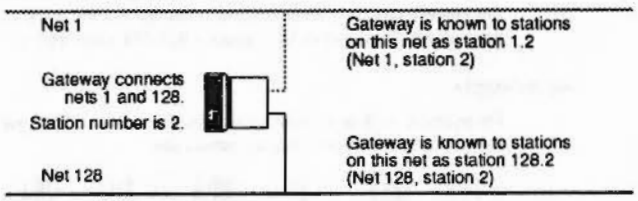
Station numbers

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. However, for convenience we recommend that you give each station a number that is unique within your site. You will then be able to move stations from one net to another without station numbers ever clashing. For example, if you have fewer than ten nets on your site you might organise your numbering scheme so that client station numbers on each net increment in units of ten, each net starting from a different base number:

Net number	Station numbers
1	10, 20, 30, 40, 50, 60, 70, 80 etc.
2	11, 21, 31, 41, 51, 61, 71, 81 etc.
3	12, 22, 32, 42, 52, 62, 72, 82 etc.
128	13, 23, 33, 43, 53, 63, 73, 83 etc.

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. One way to ensure this is to reserve a range of otherwise unused numbers for use by gateway stations. For example, in the above scheme numbers 2 - 9 are free and so could be used for gateways.

Coexistence with existing machines

This section tells you how you can use all your existing machines with an AUN network, and which machines can interact with which.

You should wherever possible use the AUN software with your existing machines. An AUN-configured machine gives you access to the greatest possible number of other machines, because on a correctly set up AUN site network:

1 All AUN-configured stations can interact with each other, no matter where they are on the site network.

Note that stations connected to Ethernet must be configured with AUN software.

You may have existing machines that cannot be configured for AUN, such as BBC and Master 128 computers, Level 3 FileServers, MDFS FileServers and FileStores. This still does not prevent you from connecting them to an Econet which is being used as part of an AUN site network. Indeed:

2 All stations connected to the same Econet network can interact with each other, whether or not they are configured for AUN.

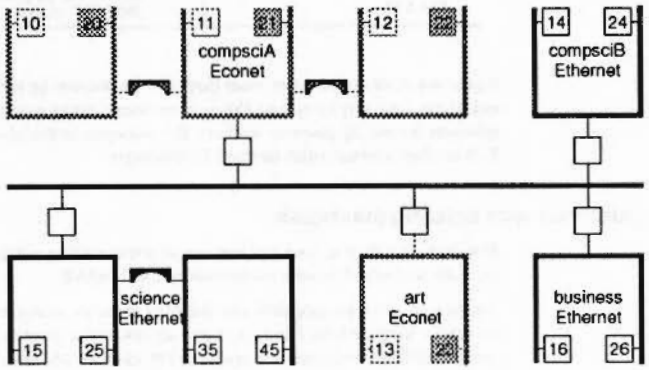
However, any stations that are not configured for AUN do not 'speak' AUN, and so are ignored by the gateway stations that connect together AUN networks. This means that:

3 Stations that are not AUN-configured cannot interact with stations on other networks – i.e. with stations on the other side of a gateway.

In particular, this has implications for file servers that cannot be configured for AUN (such as Level 3 FileServers, MDFS FileServers and FileStores). **These older file servers can only be accessed from the Econet network to which they are connected.** They cannot be accessed from the other side of a gateway.

An example

For example look at the site below, where stations 20 - 23 (shown shaded below) are not AUN-configured, but all others are:



All the AUN-configured stations can communicate with each other (rule 1 above). Let's look in more detail at the stations that are not AUN-configured:

- Stations 20 - 22 can communicate with stations 10 - 12 – and vice versa – because these stations are all on the same Econet network (rule 2). However, stations 20 - 22 cannot communicate with any other stations, because they are all on the other side of a gateway, and so are on other networks (rule 3).
- Likewise stations 23 and 13 can communicate with each other (rule 2), but station 23 cannot communicate with any other stations (rule 3).

Using redundant Econet interfaces

Stations that do not have their Econet interface configured for AUN may still use it as a native Econet interface. However, there is a restriction: you can do this either for client stations, or for file servers, but not for both at the same time.

For example, a client station with an AUN-configured Ethernet interface may also have a non-AUN-configured Econet interface, so that it can continue to access existing FileStores on an adjacent Econet. Alternatively a file server may have both types of interface: you could then access it from BBC machines over an existing Econet, while RISC OS computers access it via AUN over Ethernet.

This facility may help you to smooth any transition from Econet to Ethernet.

Coexistence with TCP/IP

Adding AUN-configured stations to an existing TCP/IP network

Because both TCP/IP protocols and AUN protocols are founded upon the same Internet family of protocols, you can use both over the same physical network. This means that you can connect AUN-configured stations to the same cabling as is used for an existing TCP/IP network, such as a campus-wide Ethernet to which UNIX workstations are connected.

The AUN-configured stations will be able to communicate with each other. If you install Acorn's TCP/IP Protocol Suite (Release 2) on these stations they will also be able to communicate with the TCP/IP stations. (The TCP/IP Protocol Suite (Release 2) is available from your Acorn Network Dealer.)

Using AUN and the TCP/IP Protocol Suite (Release 2) together is described in an application note, available from Acorn Customer Services.

Adding NFS file servers to an existing AUN network

Just as you can add AUN-configured stations to a network set up for TCP/IP, so you can do the reverse. For example you might wish to add a UNIX workstation such as an NFS file server to an existing AUN network. You can then use the TCP/IP Protocol Suite to access the file server.

Again you'll find full details in the application note, available from Acorn Customer Services.

The Broadcast Loader

In a school environment, the worst network overloading typically occurs at the start of a lesson, when a large number of clients might simultaneously attempt to load the same application from a server. The Broadcast Loader module alleviates this situation (and similar ones) by recognising that multiple stations have requested the same data, allowing the simultaneous transfer of identical data to a large number of clients from a single station. To ensure that the maximum number of clients can participate, it chops up network packets into smaller ones

Econet-based stations

The Broadcast Loader works over an AUN-configured Econet just as it does over native Econet. If you're not already using it for your Econet-based stations, you might like to consider doing so. The software is supplied as a standard part of versions of RISC OS required to run AUN – i.e. RISC OS 3 (version 3.10) or later.

Ethernet-based stations

The Broadcast Loader is designed specifically to enhance the performance of Econet-based stations. Because of the way it divides large packets into smaller ones, it will actually degrade the performance of an Ethernet-based station. We therefore strongly recommend that you **don't use the Broadcast Loader over Ethernet**.

Anyway, the higher speed of Ethernet means that you are much less likely to get network overloading, and so do not need mechanisms such as the Broadcast Loader.

Enabling or disabling the Broadcast Loader

In RISC OS 3 the Broadcast Loader is enabled by default. To disable it (or to re-enable it) use the Configure application; see the RISC OS *User Guide* for further details.

Socklib

This section describes the calls available in Socklib – the socket-level library.

We've deliberately kept this documentation 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 calls in part 2 of a 4.3 BSD UNIX manual, with the same titles as we've used.

SWI equivalents

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. You may wish to issue the SWIs yourself – say if you're programming in assembler – and the section below tells you how they correspond to the Socklib calls detailed in the following pages.

SWI names and numbers

The table below shows the title of each 'man page' that follows, the various Socklib calls each details, and the name and number of each corresponding SWI:

Title	Socklib call	SWI name	SWI no
ACCEPT	accept	Socket_Accept	&41203
BIND	bind	Socket_Bind	&41201
CONNECT	connect	Socket_Connect	&41204
GETPEERNAME	getpeername	Socket_Getpeername	&4120E
GETSOCKNAME	getsockname	Socket_Getsockname	&4120F
GETSOCKOPT	getsockopt	Socket_Getsockopt	&4120D
	setsockopt	Socket_Setsockopt	&4120C
LISTEN	listen	Socket_Listen	&41202
RECV	recv	Socket_Recv	&41205
	recvfrom	Socket_Recvfrom	&41206
	recvmsg	Socket_Recvmsg	&41207
SELECT	select	Socket_Select	&41211
SEND	send	Socket_Send	&41208
	sendto	Socket_Sendto	&41209
	sendmsg	Socket_Sendmsg	&4120A
SOCKET	socket	Socket_Creat	&41200

Title	Socklib call	SWI name	SWI no
SOCKETCLOSE	socketclose	Socket_Close	&41210
SOCKETIOCTL	socketioctl	Socket_Ioctl	&41212

Passing parameters

The parameters from the corresponding Socklib call are passed to the SWI in registers R0 upwards: the first parameter in R0, the second in R1, and so on. So for the `accept` procedure:

```
ns = accept(s, addr, addrlen)
```

the parameter `s` would be passed in R0, `addr` in R1, and `addrlen` in R2.

Any returned value is passed back in R0: so in the case of `accept`, the value of `ns` is returned in R0. However errors are not indicated by returning a specific value (unlike the Socklib calls). Instead they are indicated in the standard way used by RISC OS, described below.

Errors

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. R0 points to an error block, the first word of which contains an error number. The rest of the error block consists of a null-terminated error message.

ACCEPT

Name

accept – accept a connection on a socket

Synopsis

```
ns = accept(s, addr, addrlen)
int ns, 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, *ns*, 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, *ns*, 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.

The call returns –1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

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.

BIND

Name

bind – bind a name to a socket

Synopsis

```
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.

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*.

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.

CONNECT

Name

connect – initiate a connection on a socket

Synopsis

```
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.

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*.

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.

GETPEERNAME

Name

getpeername – get name of connected peer

Synopsis

```
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.

A 0 is returned if the call succeeds, –1 if it fails.

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.

GETSOCKNAME

Name

getsockname – get socket name

Synopsis

```
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).

A 0 is returned if the call succeeds, –1 if it fails.

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.

GETSOCKOPT

Name

getsockopt, setsockopt – get and set options on sockets

Synopsis

```
getsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *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.

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. Options at different protocol levels vary in format.

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, 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. 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.

A 0 is returned if the call succeeds, **–1** if it fails.

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.

LISTEN

Name

listen – listen for connections on a socket

Synopsis

```
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 or SOCK_SEQPACKET.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. This is currently limited to 5. 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.

A 0 return value indicates success; –1 indicates an error.

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*.

RECV

Name

recv, recvfrom, recvmsg – receive a message from a socket

Synopsis

```
cc = recv(s, buf, len, flags)
int cc, s;
char *buf;
int len, flags;

cc = recvfrom(s, buf, len, flags, from, fromlen)
int cc, s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

cc = recvmsg(s, msg, flags)
int cc, 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.

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking in which case a *cc* of –1 is returned with the external variable errno set to EWOULDBLOCK.

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:

```
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. 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.

These calls return the number of bytes received, or −1 if an error occurred.

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.

SELECT

Name

select – synchronous socket I/O multiplexing

Synopsis

```
nfound = select (nfds,readfds,writefds,exceptfds,timeout)
int nfound, 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; i.e. 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.

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.

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.

SEND

Name

send, sendto, sendmsg – send a message from a socket

Synopsis

```
cc = send(s, msg, len, flag)
int cc, s;
char *msg;
int len, flag;

cc = sendto(s, msg, len, flag, to, tolen)
int cc, s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;

cc = sendmsg(s, msg, flag)
int cc, 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 *flag* parameter may be set to MSG_OOB (otherwise 0) to send 'out-of-band' data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support 'out-of-band' data.

See *recv* for a description of the *msghdr* structure.

The call returns the number of characters sent, or –1 if an error occurred.

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.

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.

A 0 return value indicates success; –1 indicates an error.

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.

SOCKET

Name

socket – create an endpoint for communication

Synopsis

```
s = socket(domain, type, protocol)
int s, 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.

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).

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*. *Setsockopt* and *getsockopt* are used to set and get options, respectively.

A –1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

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.
[EACCESS]	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.

SOCKETCLOSE

Name

socketclose – close an open socket

Synopsis

```
socketclose(s)
int s;
```

Description

Socketclose closes an open socket, and releases any resources, including queued data, associated with it.

If an application terminates under RISC OS without closing an open socket, then that socket will remain open indefinitely.

The call will fail if:

[EBADF] *s* is not a valid descriptor.

SOCKETIOCTL

Name

socketioctl – control an open socket

Synopsis

```
#include "ioctl.h"

socketioctl(s, request, argp)
int s;
unsigned long request;
char *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 *request* are located in the header file *ioctl.h*.

The call will fail if:

[EBADF] *s* is not a valid descriptor.

AUN Driver Control Interface

This chapter describes the interface between a protocol module and a device driver module used by the protocol. The interface will enable multiple protocol stacks to control multiple different device drivers simulataneously, if required.

The interface is optimised in its detail to handle device drivers for Ethernet. Drivers for other types of network will need to emulate Ethernet in these details at this interface and map 'virtual Ethernet' values into real values meaningful to the actual connected network.

Specifically:

- physical network addresses are 48 bit quantities.
- the values of physical network frames 'owned' by protocol modules are expressed as Ethernet frame type values. For example:

  Internet owns three types of physical frame, namely frames containing IP, ARP and Reverse ARP protocol messages. The driver module will be informed of the values &800, &806 and &8035 respectively.
- At startup, driver modules must set the variable

  InetSEtherType

  to the text string name of the controlled physical interface type (et, en, etc.), with the suffix '0' (e.g. et0, en0, etc.). This is for compatibility with Acorn AUN and TCP/IP software. The name part is the same string referred to in the Driver Information Block, described later.

Service calls

When loaded, a network interface driver module will perform various internal and hardware initialisation functions. It will then announce its presence via the service call Service_NetworkDriverStatus (0) and wait for a protocol module, such as Internet, to search for device drivers controlling interfaces which the protocol wishes to access. This is done via the service call Service_FindNetworkDriver. Subsequently, if a protocol module terminates it will notify associated driver modules via Service_ProtocolDying. A terminating driver module issues Service_NetworkDriverStatus (1).

Service_FindNetworkDriver (Service Call &84)

On entry

R1 = &84 (reason code)
R2 = pointer to name of driver sought (et, en, etc.), or zero, or -1
R3 = pointer to *Protocol Information Block* describing this protocol, or zero
R4 = slot number, if R2 = -1 and R3 = 0

On exit

R1 = 0
R2 preserved
R3 = pointer to *Driver Information Block* describing available driver

Use

Service_FindNetworkDriver makes a logical connection between a protocol module and a driver module, enabling information about each other to be exchanged.

Service_FindNetworkDriver may also be used to find out whether a driver module is present, without making a logical connection. In this case R2 and R3 are zero on entry. Exit register values are the same if claimed by a driver module.

Service_FindNetworkDriver may also be used to find out which driver module controls the device located in a given backplane slot. In this case R2 is -1, R3 is zero and R4 contains the slot number (0 - 3) on entry. Exit register values are the same if claimed by a driver module.

Service_ProtocolDying (Service Call &83)

On entry

R1 = &83 (reason code)
R2 = ID of exiting protocol

On exit

All registers must be preserved

Use

Service_ProtocolDying is issued by a protocol module to notify driver modules that the protocol is exiting. The protocol ID is the same value as previously passed to the driver in pib.pib_sccall. Driver modules must never claim this service call.

Service_NetworkDriverStatus (Service Call &8B)

On entry

R1 = &8B (reason code)
R2 = status (0 ⇒ starting, 1 ⇒ terminating)
R3 = pointer to *Driver Information Block* describing this driver

On exit

All registers must be preserved

Use

Service_NetworkDriverStatus is issued by a network driver module to indicate that it is starting up or terminating. This service call should not be claimed.

Protocol Information Block

```
struct pib {
    char          pib_frtypecnt;
    unsigned short pib_frtype[6];
    int           pib_rxevent;
    struct mbuf **pib_freeq;
    int           pib_sccall;
    struct mbuf **pib_lfreeq;
};
```

pib_frtypecnt

Number of valid fields in pib_frtype[].

pib_frtype[]

Array of physical frame type values which are 'owned' by this protocol. The driver module will route all incoming frames with these types to this module, via an event sequence governed by pib_rxevent.

If a given value is zero, then all ISO format incoming frames will be routed to this protocol - in other words frames having a length field (range 64 to 1500) in place of a type field.

If a given value is 65535 (0xffff) then any packet with a frame type not matched by any other protocol will be routed to this protocol.

pib_rxevent

Number of RISC OS event to generate when an incoming frame of the correct type is received. This mechanism is used to pass incoming frames into the correct protocol module.

pib_freeq

Address of free list of 'small' data buffers owned by this protocol module but available to the driver module to store data associated with incoming frames of the correct type. [Note: driver modules must never themselves free data buffers obtained from this list.]

pib_sccall

ID for this protocol which will be included in Service_ProtocolDying on module termination. Internet = 1.

pib_lfreeq

Address of free list of 'large' data buffers owned by this protocol module but available to the driver module to store data associated with incoming frames of the correct type. These buffers should be large enough to accommodate a full Ethernet frame. There may only be a small number available so a driver module should be prepared to 'fall back' on small buffers if no large buffer is available. [Note: driver modules must never themselves free data buffers obtained from this list.]

Driver Information Block

```
struct dib {
    char    *dib_name;
    int     dib_units;
    int     dib_swibase;
    char    *dib_address[4];
    char    *dib_module;
};
```

dib_name

Pointer to 2 byte text string name of physical interface type controlled by this driver module ('ea', 'en', 'ec', etc).

dib_units

Number of accessible physical interfaces present of the type controlled by this driver module.

dib_swibase

Base of SWI block owned by this driver module.

dib_address[]

Pointers to physical addresses of interface cards. Each address is a 48 bit (6 byte) quantity.

dib_module

Pointer to title of driver module (e.g 'Ether3')

Once the startup sequence is complete, the protocol module will communicate with the driver module via SWI calls, and the driver module will interrupt the protocol module with events to indicate received frames.

SWI calls

The following set of SWI calls enable a protocol module to pass data and control commands to a device driver module. Each different driver will own a unique chunk of SWI numbers whose base is passed to a protocol module at startup time via the Driver Information Block. SWI numbers offset sequentially from the SWI chunk base will correspond functionally to the commands described below.

*Interface*_Start (Offset 0)

Start a physical interface unit controlled by the owner of dib_swibase.

On entry

R0 = unit number (0 - 3)

On exit

R0 preserved

Use

Called by protocol module to start interface and enable subsequent I/O.

*Interface*_Up (Offset 1)

Restart a physical interface unit.

On entry

R0 = unit number (0 - 3)

On exit

R0 preserved

Use

Called by protocol module to restart interface and reenable subsequent I/O, after an earlier call of SWI_NetworkIfDown.

*Interface*_Down (Offset 2)

Disable a physical interface unit.

On entry

R0 = unit number (0 - 3)

On exit

R0 preserved

Use

Called by protocol module to disable indicated interface and disallow subsequent I/O.

*Interface*_Send (Offset 3)

Transmit data via a physical interface unit.

On entry

R1 = unit number (0 - 3)
R2 = frame type
R3 = pointer to destination physical address; 48 bit (6 byte) quantity
R4 = pointer to message buffer chain containing transmit data
R5 = event number to call on completion (or zero for no event required)

On exit

R1 - R5 preserved

Use

Called by protocol module to transmit data, held in a chain of buffers. The destination physical address and a frame type value identifying the sending protocol are specified. If the protocol module wishes to be notified via an event about the status of the transmission (beyond any error value which may be passed back directly on SWI exit) then the event number (> 0) will be specified.

If R5 = 0 then the protocol module may assume that it can free the associated mbuf chain immediately on return from the SWI call, otherwise it must wait for the event to occur before freeing the mbufs.

See SWITxEventRequired below.

*Interface*_Version (Offset 4)

Return version number of DCI specification implemented

On entry

—

On exit

R0 = version number integer (this version = 3)

Use

Called by protocol module to ensure that a device driver module implements the version of DCI interface compatible with itself.

*Interface*_MTU (Offset 5)

Return physical MTU of supported network

On entry

—

On exit

R0 = MTU, or zero

Use

Called by protocol module to find out the MTU of the underlying network, for example to enable efficient fragmentation of datagrams. Default (R0 = 0) is ETHERNET MTU (1500 octets).

*Interface*_TxEvReq (Offset 6)

Return whether device driver requires an event value on TX

On entry

—

On exit

R0 = 1 if tx event number required, or 0 if tx event number not required

Use

Called by protocol module to find out whether the transmission strategy used by the device driver module requires the protocol module to supply a txevent value with every transmission SWI call (SWI_NetworkIfSend): i.e. transmission synchronous to SWI_NetworkIfSend cannot be guaranteed on request. (n.b. This may impact on performance optimisations within the protocol module.)

*Interface*_StaNumReq (Offset 7)

Return whether physical network necessarily requires a fixed station number

On entry

—

On exit

R0 = 1 if fixed station number required, or 0 if fixed station number not required

Use

Called by AUN software to find out whether the underlying network necessarily requires a fixed 'pseudo-Econet' station number (i.e. set in CMOS RAM), or whether a dynamic station number allocation mechanism can be employed by AUN software. For example, physical Econet necessarily requires a fixed station number, but Ethernet does not.

Events

An event may be generated by a driver module to indicate that a data transmission request has been processed or that a frame has been received from the network. Different event numbers are owned by different protocol modules, and are supplied to driver modules via SWI_NetworkIfSend (for tx) and Service_FindNetworkDriver (for rx) calls.

TX Event

On entry to event handler

R0 = tx event number (specified by protocol module)
R1 = pointer to data buffer chain containing tx data
R2 = pointer to name of interface controlled by this driver ('ea', 'en', etc)
R3 = physical unit number (0 - 3)
R4 = error number (driver specific) or zero = ok

A transmission event does not necessarily imply that a frame has been successfully transmitted and received by the target host, merely that the local operation has been completed - either with or without a detected hardware error - and so the protocol module may free the addressed message buffer chain. A protocol module has the option of requesting an event or no event with each SWI call to transmit data (see above).

RX Event

On entry to event handler

R0 = rx event number (protocol specified)
R1 = pointer to data buffer chain containing rx data
R2 = pointer to name of interface controlled by this driver ('ea', 'en', etc)
R3 = physical unit number (0 - 3)
R4 = rx frame type

A receive event means that an incoming frame 'addressed' (via the frame type field) to a protocol module has been received and stored within the addressed data buffers obtained for this purpose from the protocol module's freelist. Once the event is generated, the driver module must forget about the associated data buffers. These will be received by the protocol module's event handler and in due course returned by the protocol module to its own freelist. Data buffers comprising an individual frame are chained together via the m_next field (see below).

The first buffer in each frame chain does not contain frame data. The first four bytes contains a pointer to a Driver Information Block describing this driver. The next six bytes contain the 48-bit physical address of the source of the received frame.

The rx frame type (R4) should be set to 0 to indicate that the protocol module should discard this mbuf chain immediately, for any reason.

Data buffers

Data passes across the interface between the protocol and driver modules in 'mbufs'. These are similar to the data structure as used internally within the BSD UNIX kernel, and also within the RISC OS Internet module, for handling network data. Mbufs are aligned on 128 byte boundaries and can either store internally up to 112 data bytes, or else reference an external block of data. The format is:

```
#define MSIZE           128
#define MMINOFF         12
#define MTAIL           4
#define MLEN            (MSIZE-MMINOFF-MTAIL)

struct mbuf {
        struct  mbuf *m_next;           /* next buffer in chain */
        unsigned long  m_off;           /* offset of data */
        short   m_len;                  /* amount of data */
        char    m_type;                 /* mbuf type */
        char    m_indir;                /* data is indirect */
        union {
                char  mun_dat[MLEN];    /* data storage */
                char   *mun_datp;       /* indirect data pointer */
        } m_un;
        struct  mbuf *m_act;            /* used by protocol module */
};
#define m_dat    m_un.mun_dat
#define m_datp   m_un.mun_datp
```

If the data is indirect then the device driver module must make no assumptions about the actual location of the referenced data.

12 AUN *Commands

5

This chapter gives details of the * Commands provided by the AUN software. These commands may help you in managing your network, and seeing how it is operating. To use the more esoteric commands you will need a more technical understanding of AUN than we have so far given you, particularly its use of Internet addressing.

The list below summarises the commands in this chapter:

*Configure BootNet

Sets the configured state for whether or not the AUN software is loaded from ROM

Syntax

```
*Configure BootNet On|Off
```

Use

*Configure BootNet sets the configured state for whether or not the AUN software is to be loaded from ROM. Drivers are always loaded from the ROM, irrespective of this configured setting. **This command is only available on stations fitted with an AUN client ROM.**

For such stations, 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 at installation of the card is 'Off'.

Example

```
*Configure BootNet On
```

Related commands

None

*DeviceInfo

Displays driver module internal statistics

Syntax

```
*EcInfo
*E1Info
*E2Info
*E3Info
```

Use

A *DeviceInfo command displays detailed information about driver module activity. Each of the standard Acorn drivers provides such a command:

Command	driver for:
*EcInfo	Econet
*E1Info	Acorn Ethernet I
*E2Info	Acorn Ethernet II
*E3Info	Acorn Ethernet III

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

```
*E3Info
Ether3 interface statistics

ea0: bussize 16 (1a), slot 0, enabled, hardware address 00:02:07:00:79:00

packets received = 27735              packets transmitted = 2391
bytes received = 2040394              bytes transmitted = 392460
receive interrupts = 27279            transmit interrupts = 2390
interrupts = 29658

Frame types recognised: 0x0800, 0x0806, 0x8035.
```

Related commands

None

*InetInfo

Displays Internet module internal statistics

Syntax

```
*InetInfo
```

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

Resource usage:

Sockets
    Active 10
Data buffers
    Total 512, InUse 0, Hiwat 40, Mfree 238, Mfreerx 234, Mfreerxl 18

Packet forwarding not in operation
```

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         EconetA                information for first AUN interface
AUN Station       4.5
Full address      1.4.4.5

Interface         Ether2                 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
art             gateway=131.5
backbone        local
```

a client on the 'backbone' net connected to the 'art' net by gateway 131.5

```
*Networks
art             local
backbone        local
```

a gateway between the 'art' net and the 'backbone' net (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 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

13 AUN Technical information

5

This chapter gives some more technical information on how the AUN software works. You don't need to read this chapter, since you can install, use and manage the network without knowing any of the information it contains. However, the more technically minded amongst you may be interested in what follows.

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 three closely related modules:

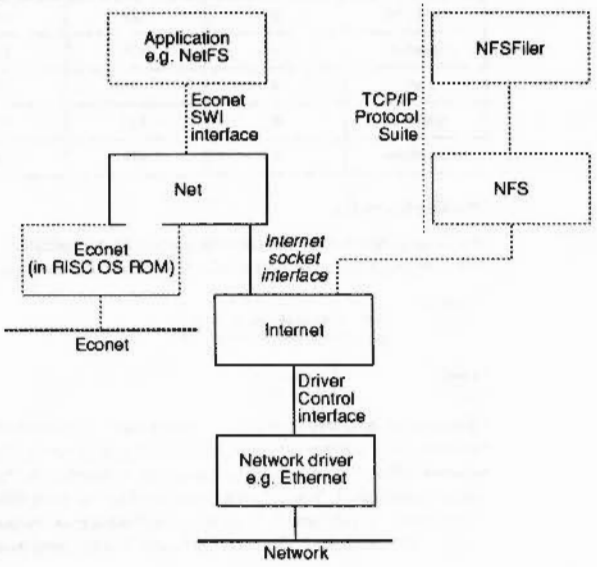
- The **Net** module implements the two-way acknowledgement handshake, and presents an Econet-like service to applications via Econet SWI calls. It also implements the RIP function.
- The **Internet** module implements UDP, IP, ARP and RevARP protocols, and exports an industry standard (Berkeley socket) interface to other RISC OS software such as the TCP/IP Protocol Suite.

device driver.

The AUN software comes with several driver modules: **EconetA** (for Econet interfaces), **Ether1** (for an Acorn Ethernet I card), **Ether2** (for an Acorn Ethernet II card) and **Ether3** (for an Acorn Ethernet III card). The Econet driver accesses the network interface via the Econet handler resident in the RISC OS ROM, whereas the Ethernet drivers directly access the Ethernet hardware.

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.

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)

To find the site, network and net numbers of both its interfaces, the gateway station looks at its Map file and its Configure file.

The Map file

The Map file tells the gateway station the IP address of each net on the site.

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 x 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 x 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 x 100) / 50 centisecond timeouts;

Else

If the retry delay < 25 centiseconds

exactly 50 retransmissions will occur;

Else

(T x 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.

14 User Interface

5

Overview

This chapter describes the changes made to the window manager, Filer, Pinboard and DragASprite. These changes fall into four main areas:

- Outline system font
- Improved error handling
- Sprite tiled windows
- New memory management

Desktop appearence

There have been sprite and template changes to give a 3D appearence 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 and Pinboard have been modified to take account of these changes. DragASprite has been modified to use dithered solid drags so that you can see the area underneath solid drag.

New Error system

The WIMP error messages have been changed to be more helpful, consistent and user friendly. Applications can now provide a more suitable wording on Error messages and buttons.

Concepts and definitions

Text in Icons

Currently, for plotting text in icons, the wimp calls the kernel using the OS_Write and VDU calls, we shall refer to this method as 'VDU text'. This provides a bit-mapped fixed width font which is generally referred to as the system font. In the new system the wimp will call the font manager, allowing the use of a proportionally spaced font. We expect that the outline font system will be used most by high-end users, and by those who possess high-resolution (square pixel) monitors (referred to as VGA resolution from now on). However, we do not want to

exclude those with low (TV) resolution monitors, and wish to provide them with a reasonable path forward. The basic approach is that we will continue to have a single font at any one time which is used for most purposes within the window system. (This contrasts with Windows, for instance, where numerous different fonts are used for different purposes). For the purposes of laying out dialogue boxes etc. applications developers should be aware that this font may be one of the old system font (VDU text), Homerton.Medium or possibly Trinity.Medium (both at 12-point). We expect VGA-resolution monitors to be used with the normal anti-aliased font as rendered by the font manager. We expect TV-resolution monitors to be used with a hand-tuned bitmap at 12 point, this will be referred to as Darwin.Medium .

The Error System

A RISC OS error is a block containing a unique error number for that error, and a textual string. Programs can identify a particular error by examining the unique number. The human-readable text is translated into the users language, when the error block is created. It's also possible that an application may wish to display infomation rather than an actual error using Wimp_ReportError so we refer to these collectively as a Report.

Report Categorisation

A new scheme of report categorisation is introduced (based on setting bits of the flags-see Programmer Interface section). The different categories are: error, information, program and question, each identifiable by a different sprite in the error dialog (These are shown in the User Interface section of this document). As the PRM states (page 1-43) bit 31 of the error number is used for exceptions like data aborts, these are always interpreted as 'program reports'. Program reports are ones that a normal desktop user should never see in normal operation of his machine. They strongly imply that a program somewhere (system or application) contains a bug. The user need not know the details of the cause, although an expert user might be interested. Its quite possible that an application will have to be terminated to get out of this. Its even possible that the machine is no longer usable until a reset.

Non-program 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 (if they happen) do in fact have to be understood by the user. These include: Incorrectuser input Running out of a resource such as memory or disc Disc sector error - or, other non functioning hardware

Of running reports, an error indicates that something serious/unfortunate has happened, even though it might not be a programs fault. Examples include malfunctioning hardware, disc corruption, a corrupt data file, resource file

missing. An information report is more an information bulletin than an error. No evasive action is typically required of the user. It can be used for confirmation of some important activity. A 'question' is desirable when the message is neither information nor an error. For instance this might be used when the user is trying to quit with unsaved data. If the text is along the lines of 'There is unsaved data, do you wish to discard, save it or cancel quitting?' then confusion should be reduced by using the question sprite rather than the error or information.

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. A watchdog will be added to the wimp so that such rogue programs may be killed and increase the chances of the machine remaining usable without the need to reboot.

User Interface

Using an outline system font

The !Configure application will be extended to allow easy choosing of a desktop font. Since dialog boxes will only look tidy for a small range of font sizes (see Appendix A) !Configure will choose a size that will work with most dialog boxes. For the softloaded WIMP to be released for RISC OS 3.1x though (see section on product organisation), *Configure will be used as described in the Programmer Interface section. If painting generates an error for any reason, then the wimp does not report an error but falls back to the system font. (This is particularly important when painting the error box itself).

New error system

The Wimp error box is replaced as follows (note the use of the outline font and background):

The title bar contains either Message from Appname or just Appname (replacing the previous Error from). Sprite1 is usually the sprite for the application generating the error, or a warning symbol if this is not known. Sprite2 is one of error, program, question or infromation as shown below:

Button1 above is a default action button, and typing RETURN selects it. The standard buttons are labelled Continue and Cancel (Button 2), but the application can arrange for any button wording. If the buttons make the window too wide (which should be rare) then the window increases in width to accommodate them.

The error text is either the error message supplied by the application, or, in the case of a program error, replaced with 'Appname has gone wrong, click Quit to stop Appname'.

Tiled window backdrops

If the wimp finds a tile_1 sprite in either a windows sprite area or its own (eg. loaded with *Iconsprites) then this is used to tile the backdrop of a window with background colour 1. The new style error dialog box below has such a background.

The watchdog

This is intended to kill off a task at the request of the user, though typically it should be used as a last resort. Choosing 'Quit' from the application's iconbar menu or the task manager is obviously preferred as this enables the task to do any memory deallocation. If the user presses Break in the desktop, this leads to an dialog box Press Stop to terminate foo with buttons Stop and Cancel and Next task. Clicking the last of these, changes the question to refer to the next task in the task list. This should work even if Wimp_Poll is not being called, eg. for killing off rogue redraws, looping programs and the like. This also works when the Wimp_ReportError box is up, eg. to kill off a task which is looping in redraw, producing what it thinks is a user error.

The Filer

There are five main main changes to the filer's user interface:

- variable column width in the filer viewers
- use of outline font for size/date etc. in full info mode
- no longer a limit of ten characters to a filename
- use of solid dragging
- displaying of open directories

As outline fonts typically have larger capitals than the VDU text of a similar size, there is a possibility that the fixed column width (defined in the Filer's templates file) may be too small. As a result the filer will now check the lengths of all the filenames in a displayed viewer and alter the width accordingly as shown in the examples below.

The Filer currently uses VDU text to print the information in 'Full Info' mode. This will be changed so that the current desktop font is used for the date and size etc.

Filesystems, such as NFS, support filenames longer than ten characters. These are currently truncated by the filer and only the first ten characters are displayed. With the variable column width system described above it will be possible to display the full filename, though a limit of 63 characters is imposed for practical and ergonomic reasons- the viewer may become unusable for instance if there is one extremely long filename in it.

The filer will also use solid dragging (see picture in DragASprite section below) and when multiple files are being dragged a 'package' sprite will be used instead of the outline.

Open directories will become more easily identifiable by displaying the directory folder sprite as being open.

The Pinboard

The only change required to the pinboard is that when displaying an icon on the back window, the text underneath is no longer constant in width.

DragASprite

The changes to DragASprite are unrelated to the outline font system. As shown below solid drags will be 'dithered' so that it is possible to determine what is being dragged over or dropped on.

Programmer Interface

This section describes the programmer interface of the new features. It is divided into the following sub-sections:

- Outline system font - programmer and system notes for the desktop font
- Changed SWIs - details of existing SWIs which have been modified
- New SWIs - details of SWI calls which are totally new
- Service calls - new service call mechanisms
- * Commands - new CLI commands

Outline system font

Controlling the desktop font

CMOS byte &8C has the following interpretation:

bit 0	2D/3D switch as described below	
bits 1-4:	value	meaning
	0	use Wimp$Font, Wimp$FontSize,Wimp$FontWidth as described below
	1	use VDU text as system font.
	2-15	use ROM font (as enumerated in Resources:$.font) at 12 point

bits 5-6	reserved (must be zero)
bit 7	if set then desktop tiling is disabled.

The ROM fonts are enumerated in the following way: Starting at Resources:S.Fonts, every directory which has a descendant IntMetric* (eg. IntMetrics, IntMetric0) is included in the list. Eg. on a standard system the following mappings would occur (though it should be noted that a certain mapping can not be assumed):

2-Corpus.Bold	3-Corpus.Bold.Oblique	4-Corpus.Medium
5-Corpus.Medium.Oblique	6-Darwin.Medium	7-Homerton.Bold
8-Homerton.Bold.Oblique	9-Homerton.Medium	10-Homerton.Medium.Oblique
11-Trinity.Bold	12-Trinity.Bold.Italic	13-Trinity.Medium
14-Trinity.Medium.Italic	15-WIMPSymbol	

A new configure option is also added:

*Configure WimpFont <number> where <number> can be 0-15 representing one of the actions as bits 1-4 above.

When !Configure does not indicate the use of a ROM font (bits 1-4 being 0) the variables below define the outline font to use with the wimp:

Wimp$Font	the font to use
Wimp$FontSize	the size (height) of the font in 1/16ths of a point
Wimp$FontWidth	the width of the font

The width is optional, if omitted, the font size is used. If the size is undefined, a value of 192 (12 point- approx the size of the system font) is used.

eg.

To set Corpus.bold in 12*10 point (ie narrow)

```
Set WimpSFont  Corpus.bold
Set WimpSFontWidth 160
```

as with all system variables, capitals are optional.

3D look

It is not required that an application be able to function in both 2D and 3D. Applications that provide this facility should decide which to present by looking at bit 0 of CMOS byte &8C.

1 = 3D desktop look
other = 2D desktop look

This should be rechecked whenever there is a mode change.

Message_FontChanged

The values of the variables and CMOS settings described above are checked at desktop startup, at a mode change, and when the broadcast Message_FontChanged is sent by any application:

when broadcast as a user message:

```
mblock+16      Message_FontChanged (&400CF)
```

On receiving this the Wimp loses the current font, examines the variables as defined above and attempts to find the defined font with the font manager. If this happens successfully then the Wimp broadcasts the following message to all tasks:

```
mblock+16      Message_FontChanged
mblock+20      Font handle or 0
```

then all the open windows are redrawn. This is necessary so that old applications which do not understand this message will appear correctly. If the font could not be found (eg. because wimpSfont was unset) then the system reverts to the VDU text system. The font handle is used in the filer for instance so that it can calculate how wide the directory viewer columns should be (see the section on external dependencies).

Fixed width icons

Some applications find it necessary to display an icon on the iconbar and then change the text under the icon to relate the status of the application whilst running. Some programs, such as !Printers, create and delete their icons when the text changes, whilst others (eg. !Teletext, !BB) create their icon with a fixed width and just update the text at run time. Currently the application assumes that the widest the icon needs to be is 16 OS units times the number of characters in the longest string.

This will no longer work. An application must measure the size of all potential strings using the SWI Wimp_TextOp defined below. These should preferably be cached and recached on Message_FontChanged. The icon should then be created so that all strings (and the sprite) would fit. On a Message_FontChanged, the

Wimp will do its best to resize iconbar icons but obviously in this case the application knows better. The program may either call Wimp_ResizeIcon as defined below or delete then recreate the icon.

Miscellaneous programmer information

When using an outline font or the built-in VDU font, the wimp will perform auto-computation of menu width. The 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. Auto width calculation is also used for creating icons on the icon bar. In this way the icon need only be specified as being the width of the sprite, the wimp then calculates the required width so that all the text is readable. The width is recalculated automatically on a mode or font change.

In menus, keyboard shortcuts must be displayed right-aligned. These are detected using the following rule: If a non-writable menu entry contains at least one space, and the string after the last space starts with no more than one of the patterns in a list in the Wimps Messages file, called Modifiers, and ends with a pattern from another message file list, called KeyNames, then right-align everything after the last space. In the UK the Modifiers will consist of \x8b ^ ^\x8b \x8b^ (hex 8B is the shift key in the symbol font, '^' symbolises the control key) and the KeyNames list will consist of:

ESC PRINT INSERT DELETE COPY HOME PAGEUP PAGEDOWN TAB ENTER RETURN Esc Print Insert Delete Copy Home PageUp PageDown Tab Enter Return F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12.

In this way both the modifiers and key names are internationalisable, eg. it is possible for the modifiers to be a whole word rather than just a symbol, as we use in the UK. Application writers are encouraged to use the modifiers and keynames as listed in the style guide.

Changed SWIs

For the following existing SWIs, omission of Entry/Exit details implies that there is no change to the interface as described in the PRM.

Wimp_ReadSysInfo (&400F2)

This has the following new Reason codes (to those described in the PRM, page 3-218...) for returning information about the desktop such as the system font handle:

On entry

R0 = information item index

On exit

R0 = information value

Use

R0 on entry	On Exit
8	R0 = font handle of desktop font or zero if wimp is currently using VDU text R1 = symbol font handle or undefined if R0 = 0
9	R0 = pointer to wimp toolsprite control block

Wimp_ReportError (SWI &400DF)

At the moment, this provides the following: (see RISC OS 3 PRM, page 3-179)

On Entry

R0 = pointer to error block (code + message)
R1 = flags
R2 = pointer to application name

On Exit

R0 corrupted
R1 = 0 for no key click, 1 for OK, 2 for Cancel

Flags:

Bit	Meaning when set
0	provide OK
1	provide Cancel
2	highlight Cancel
3	immediate return from Wimp_CommandWindow text window
4	dont prefix Error from in title bar
5	return immediately with box up
6	close the box (using 0 and 1 to select a button)
7	do not beep
8-31	reserved - must be 0

The following is added to this:

If bit 8 of the flags is set then:

R1 Bits 9...11	Meaning
0	Old error sprite (non classified)
1	This is an information report.
2	This is an error report.
3	This is a program report.
4	This is a question.

R3 = pointer to sprite name
R4 = pointer to sprite area (or 1 to use the Wimp sprite area)

R5 = pointer to additional list of buttons, or 0 if none.

Use

R3-R5 are assumed valid if bit 8 is set. If no sprite name is provided (R3) the Wimp tries the application name prefixed by "!" as a sprite name (a desperate measure as it may not internationalise well). This applies to all old errors as well. R5 is a pointer to a comma-separated sequence of strings, terminated with a control character. These strings are the text of additional buttons to create in addition to the Continue and Cancel buttons (if requested). If neither Continue nor Cancel is specified by bits 0 and 1 then the first of these is the default button. These buttons will generate return values 3, 4, 5 and so on, even if button 3 (the first custom button) is the default. The first of them will appear to the right, the next to the left of that, and so on. If Describe is added by the system this appears at the extreme left.

If the report is a program error (either bit 31 OR (bit 30 AND bit 23) of error number set or combination 3 above) then a Cancel button is always provided, but the label on it is Quit rather than cancel. The error text is replaced by: Appname has gone wrong, click Quit to stop Appname. A Describe button is added to the button list. If Cancel/Quit is pressed and the program did not request its presence, then the application is simply terminated by the Wimp without being reentered. This will be done by calling the tasks exit handler (as the error handler may simply call Wimp_ReportError once more which would be confusing). If Describe is pressed then the message is replaced by the original error message and buttons provided by the application, ie. the Describe button disappears.

(Aside: this assumes the convention used by many applications, that in the case of an unexpected error Cancel is used to actually exit the program while OK is used to attempt to continue.)

DragASprite_Start (SWI &42400)

On Entry

R0 = flags

Use

A new flag bit is added (bit 8) to control dithering. If unset (as would be the case for all exisiting applications) then dithering is enabled. Setting the bit disables the feature.

New SWIs

Wimp_TextOp (&400F9)

On entry

R0
bits 0-7 Reason code - 1,2
bit 31,30 flags
bits 8-30reserved (must be zero)
R1-R7 depends on reason code

On exit

Depends on reason code, though typically R0 currupted, R1-R7 preserved

Use

This call is used by an application to manipulate and display text using the current desktop font. The Wimp calls Font_Paint or OS_Write according to whether VDU text or an outline font is currently being used in the desktop. If an outline font is in use, the string is checked for symbol characters before the operation takes place.

Wimp_StringWidth (1)

On entry

R0 = 1
R1 = string pointer (control character terminated)
R2 = character to stop at

On exit

R0 = width of string in current font in OS units

Use

This call is used to get the width of a string (normally before plotting it in an icon or using RenderText below) for the current desktop font. For instance, it will be used by the Filer when calculating the widths of the columns in the viewer. 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.

RenderText (2)

On entry

R0 = 2 (+flags)
R1 = string pointer (control character terminated)
R2 ,R3 reserved, should be -1
R4 = bottom left x-coordinate in screen OS units
R5 = bottom left y-coordinate in screen OS units

On exit

If V Set, R0 = error block

Use

This call may be used to plot 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 where it is for VDU text even when an outline font is in use. It is intended that this call be made from a redraw loop and as such Wimp_SetColours is used to determine what colours are used for the text. It should be noted that because an outline font may be used, the background colour should be set as well so that the antialiasing colours may be found. This call does not preserve the current font, nor the font colours.

Wimp_SetWatchdogState (&400FA)

On entry

R0 = state (0=disable, 1= enable)

R1 = code word or 0

On exit

V Set, R0 -> error block

Use

This call may be used to enable or disable the state of the watchdog. It is intended that it will be used 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.

Wimp_Extend (&400FB)

This call is for Acorn use only, you must not use it in your own code.'

Wimp_ResizeIcon (&400FC)

On entry

R0	Window handle (-1 for iconbar)
R1	Icon handle
R2-R5	New Icon bounding box

On exit

V Set, R0-> error block

Use

This call is for resizing icons that have already been created on a window. Although general purpose, it is most likely to be used by an application needing to resize icons after a font changed message. It will not invalidate the area of the icon, though typically this will not be required since the font changed message is followed by a redraw request. As the icon bounding box is given, this call may also be used to move a previously created icon.

Errors

Invalid window handle
Invalid Icon Handle

Service Calls

A new service call interface allows a separate module to replace the entire system or add new buttons, such as a debug button or a help button. Pressing these new buttons returns control to the module that sent them, via another service call. As a return from this call the module can specify that the ReportError SWI exits, or that the box is reentered. In reentering the box the text in the message window can also change.

When Wimp_ReportError is called, the following service is issued:

Service_ErrorStarting (Service call &400C0)

On Entry

R1 = &400C0
R2 = pointer to error Block
R3 = flags
R4 = app name
R5 = pointer to sprite name
R6 = pointer to sprite area or 1
R7 = pointer to button list or zero

On Exit

As Entry

Use

R2-R7 have the same interpretation as R0-R5 for Wimp_ReportError. The actual values of R2-R7 may be altered, but not the memory pointed to by them. By setting the flags to exit immediately (R3 = 64) a module claiming this call may handle errors in its own way. It would also need to claim Service_ErrorEnding below and set a suitable button number for return to the task. A module may add buttons by copying (R7) and appending (not inserting) its own button list. It should obviously keep track of the position of its button for use in the button pressed handler. The toolkit should make this easier to use.

Service_ErrorButtonPressed (Service call &400C1)

This occurs when any button on the error dialog box is pressed.

On Entry

R0 = 0
R1 = &400C1 (reason code)
R2 = button number (1-OK,2-cancel,3-right most custom button...)

R3 = pointer to button list as it appeared on the dialog (note that if more than one module adds a button, this may not be the same as when the initial ErrorStarting call came round)

On Exit

R0 = 0 - return to application
R2 = button number to return (normally unchanged)

OR

R0 = 1 - redisplay error dialog
R2 = pointer to new error register block:

+0	error block pointer
+4	flags
+8	title string
+12	sprite name pointer
+16	sprite area
+20	button list (or zero)

Use

If a module sets up such a block, it should probably claim the call. If not, other modules which add buttons (or change the block themselves) could lead to none of them knowing exactly what state the error box was displayed in and thus the functionality will probably not work as it was intended. It need not be claimed if only minor changes are made (eg. changing the flags to turn the beep off).

When any button on the error dialog box is pressed the following service call happens:

Service_ErrorEnding (Service call &400C2)

This call occurs when the error box is about to close

On Entry

R1 = &400C2 (reason code
R2 = button number to be returned to application

On Exit

R1 = 0 to claim call

R2 = button number to return to application.

Use

A module may claim this call to alter the button number that is returned to the task that initiated the error. This is only of real use when the module has dealt with the error itself in some way.

* Commands

*WimpKillSprite

Syntax

*WimpKillSprite sprite-name

Parameters

sprite-name, name of sprite in sprite pool (eg. small_fff)

Use

This will remove the given sprite from the wimp sprite pool. It should be used with care as deleting certain sprites will cause some applications to fail. For example the directory sprite is necessary for the Filer to work.

Example

*WimpKillSprite file_fff

Errors

Sprite doesn't exist the sprite given was not found.

15 Desktop boot configuration

Introduction

!Boot is an application which resides on the hard disc which gets invoked when the computer starts up. It's job is to set the machine up for the user whenever the computer is reset.

!Boot operates in two ways, it works automatically when the computer starts up, and it works interactively when a user double clicks on the !Boot icon. The interactive user interface is described briefly in this chapter and more thoroughly in the User Guide. This chapter mainly describes the internal working of the various boot files and directories within the !Boot application.

!Boot takes over the functionality of the following applications.

- !System, !Scrap and !Fonts are now sub-applications contained within !Boot. The end user does not directly use these applications.
- A modified version of the RISC OS 3.1 application !Configure is now a sub-application contained within !Boot.
- A modified version of the file system lock utility !FSLock is now a sub-application contained within !Boot.

Overview

!Boot controls the desktop configuration of the computer. It gives users and applications control over the start up and use of the desktop.

It allows:

- Desktop boot saving - as in RISC OS 3.1
- Hard disc locking with a password
- Computer configuration using a !Configure type application
- !System, !Scrap and !Fonts are now part of this application
- Applications can modify startup files
- Applications can be linked to the pseudo-directory Resources:$.Apps that is displayed when you click on the Apps icon.

File system locking

The file system locking module gives the ability to lock the hard disc drives on any hard disc filing system.

When the filing system is locked, any attempt to write to the filing system just displays an error message. There are however two exceptions to this.

If the computer is locked, users can still write to the special directory `$.Public`. They can also create additional directories under this directory.

Additionally applications can also write to the !Scrap application (contained in !Boot), this allows temporary application space can be created and used.

If you don't want a Public directory that can always be written to, you should delete it before locking the computer. The Public directory cannot be created on a locked computer.

Even though the filing system is write protected, users can still read the filing system and can copy data onto the Public directory or onto floppy disc.

By default the locking system only protects discs that use the internal IDE disc interface.

Technical details

User Interface - overview

The !Boot application must always reside in the root directory of the hard disc that is defined as the boot disc.

!Configure, the part of the application that is used to change the default operation of the system can be erased from the !Boot application. This make reconfiguration of the computer impossible. However if the system locking facility is used effectively this extreme course of action should not be necessary.

The reconfigure window

This is based on the RISC OS 3 !Configure application. The following areas have been changed:

The Floppies, Net, Keyboard, Memory and Sound windows remain the same as they were in RISC OS 3.1.

Printer window. This has been removed as it only confused the user.

Applications window. This has been removed. Applications are now auto-started from the desktop boot file and all application are stored on hard disc.

Discs window. The ST506 section has been removed as ST506 support has been removed from the operating system.

Mouse window. There is now a choice of mouse type, see page 5-158 for more information.

Screen window. This allows the choice of monitor by name and a choice of mode by parameters. There is also an area that will allow a choice of background screen texture.

System window. This window allows you to merge the contents of !Systems. The utility !Merge (or !SysMerge) is no longer needed.

Fonts window. The following new areas have been added. You can choose a font to replace the standard system fonts used in windows and menus. You can use any of the fonts available on your computer. This window also allows you to add new fonts to your computer's existing fonts by merging !Fonts applications.

Windows window. The following new areas have been added. You can choose to use the desktop in 2D mode rather than in the default 3D desktop mode. This window also allows you to add textured backgrounds to your windows.

Lock window. This window controls whether your computer is locked or not, and what the password is. A locked computer's hard disc cannot generally be written to.

There lock window has three field – one for the old password and 2 for the new password, and three action buttons - (Unlock/Lock), Change password and Cancel.

If the machine is locked the Unlock/Lock button is set to Unlock and if the machine is unlocked it is set to Lock. This button is the default action.

To change the password both of the new password fields must be filled in with the same value and the old password must be correct. If the new password is empty then the machine will have its password removed, thus making the machine boot in the Unlocked state. If a new password is filled in then it will replace the old. The Change password button actions this change.

Using the Lock window

- To unlock the machine double-click on !Boot, click on the lock icon (the only available icon), enter the password and press Return.
- To lock the machine double-click on !Boot, select the lock icon (all the other icons should be available at this stage) and press Return.
- To change the password select the lock icon, enter the old password, down-arrow to the next icon, enter the new password, down-arrow to the last icon, enter the new password again then click on Change Password.

Sequence of activities

This is what the start-up procedure when the computer is switched on or restarted.

Files run	Variables set	Comments
!Boot.!Run	Boot$Dir Boot$Path	
	Boot$ToBeLoaded	
	Boot$ToBeTasks	
<choose which of desktop/boot run>		
!Boot.Utils.BootRun		Obey -c used to give robustness.
!Boot.Choices.Boot. PreDesktop		Obey -c used to give robustness.
	<Aliases>	Sets up command aliases
	<Paths>	Sets up useful paths
	<Options>	Sets up program options

Files run	Variables set	Comments
	<ResApps>	Sets up links from Resouces:$.Apps to applications
<Misc>		By default empty
<Loads/Runs stuff in Boot$ToBeLoaded>		
!Boot.Choices.SetUp.		Run the desktop
Desktop		file.
Filer_Boot		
!Boot.Resources."		
Filer_Run		
<Boot$ToBeTasks>."		

Environment set up

This section lists the system variables and functions that are set up while running !Boot.

System Variables

Variables set	Comments
Boot$Dir	Standard
Boot$Path (macro)	Standard
Boot$ToBeLoaded	Points to directory of stuff to be Loaded by PreDesktop. Expected to be used by applications.
Boot$ToBeTasks	Points to directory of stuff to be WimpTasked when the desktop starts up. Expected to be used by applications
Alias$Alias	Alias to set aliases with
Alias$UnAlias	Alias to unset aliases with
Alias$Path	Alias to set paths with
Alias$PathMacro	As Alias$Path, but sets Macros
Run$Path	Set to ',%.,Boot:Library.' to allow direct running of the boot utilities

Functions:

In <Boot$ToBeLoaded> the following actions occur:

Modules – RMLoad
Sprites – *IconSprites
Obey files – Obey -c
Directories – Run

Everything in Boot:^.Apps is turned into a ResApp
Everything in Boot:Resources is acted upon by Filer_Boot
Everything in <Boot$ToBeTasks> is acted upon by Filer_Run

Parts of !Boot to be modified by applications

The contents of <Boot$ToBeLoaded> and <Boot$ToBeTasks> are expected to be modified by applications. Each application !<App> is given the entry called <App> in these directories. The application may do what it likes with this, create it as a directory or as a file. The modification of this directory/file is left up to the application in question.

The PreDesktop file

The !Boot.Choices.Boot.PreDesktop contains the command line setup sequence. It gets invoked using Obey -c so that filing system or network software can be reloaded during its execution. The file has been divided into well defined sections for ease of maintenance:

These areas are:

- Aliases
- Paths
- Options
- ResApps
- Misc

Aliases

This section sets system aliases. The first alias set is for Alias itself so that the following command works:

```
Alias <alias> <command>
```

Which sets the alias <alias> for the command <command>.

The next two aliases are for convenient setting of paths:

```
Path <path> <full path>
```

which enables <path>:rest to be used for <full path>rest. For example:

```
Path lib ADFS::HardDisc4.$.Library.
```

would enable the following convenient commands:

```
*Dir lib:
```

```
/lib:cc
```

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>.

The next aliases are commented out as they may not be of as general use:

touch - alias for Stamp
mv - alias for rename
Basic - alias for Basic64
Mode - alias to change mode

Paths

This section has been set aside for setting standard paths and directories. The only path defined here is Run$Path which has been set to include Boot:Library. The other paths are programming language specific and have been divided into three sections, one each for Basic, C and Assembler.

```
Basic:
|Path BASIC ADFS::HardDisc4.$.BasicLib.

C:
|Set CLib$Dir ADFS::HardDisc4.$.CLib
|PathMacro CLib <CLib$Dir>.
|Set RISC_OSLib$Dir ADFS::HardDisc4.$.RISC_OSLib
|PathMacro RISC_OSLib <RISC_OSLib$Dir>.
|PathMacro C <CLib$Path>,<RISC_OSLib$Path>
|SetMacro C$LibRoot <CLib$Dir>,<RISC_OSLib$Dir>

Assembler:
|Path Hdr ADFS::HardDisc4.$.Hdr.Hdr.,
ADFS::HardDisc4.$.Hdr.Hdr2.,ADFS::HardDisc4.$.Hdr.EcoHdr.
```

Options

This section has been set aside for miscellaneous options. Two examples are placed here to give people the idea:

```
|set NFS$TruncateLength 999999
|set Copy$Options "A ~C ~D F N L ~P Q R ~S ~T ~V"
```

ResApps

This section registers applications with ResourceFS for display in Resources:$.Apps.

```
AddApp Boot:^.Apps.!*
```

Misc

Other bits of setup should go here. Examples include loading and binding a novel system beep.

Internal applications used by !Boot and other applicatons

The AddApp transient utility

This is used internally within !Boot, but it is also expected to be used in a more challenging environment.

```
AddApp [<directory>.]<wildcard>
```

This will add entries into Resources:$.Apps for all applications matching the wildcard in the given directory. If the directory is not specified then the current directory will be scanned instead. It is not an error if there are no matches.

The IfExists transient utility

This is used internally by !Boot, but it is expected to be used in a more challenging environment.

```
IfExists <file> Then <true command> [Else <false command>]
```

This will OS_File 17 <file> and if it exists then <true command> will be executed, otherwise <false command> will be executed. Note that non-files (directories and partitions) will cause the true command to execute.

The Repeat program

This program is used internally within !Boot, but is expected to be used in conjunction with the AppSize transient utility for other purposes.

```
Repeat <Command> <Directory> [-Directories]
[-Applications] [-Files | -Type
<file type>] [-CommandTail <cmdtail>] [-Tasks]
```

This scans the directory <directory> applying command to everything it finds within the limits of the other parameters:

- `-Directories` - do it to directories only
- `-Applications` - do it to applications only
- `-Files` - do it to files only
- `-Type <file type>` - do it files of the given type only

The command executed is:

```
<Command> <thing found> [<cmdtail>]
```

If `-Tasks` is specified these will be WimpTasked.

This utility does not recurse. Only those objects identified at the top level have the command applied to them.

The AppSize program

This program is used to control memory before the desktop has started. It causes memory to move from application space to the RMA so that modules and stuff can be loaded.

```
AppSize <size>[K]
```

The RMA will be grown by <current app size>—<size>.The memory is first obtained from the free pool, so the application space may be a slightly bigger than requested.

The FontMerge program

This program is used to merge new fonts into an existing !Fonts directory. Its command line is:

```
FontMerge <source> [<destination>]
```

Where <source> is the directory of fonts to merge into <destination>. <destination> is optional, in which case 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$Prefix>.,R
esources:$.Fonts.
```

The last element is in Resources - impossible to merge into here. The next-to-last element in <Font$Prefix> for old style compatibility - not a good idea to try merging here. The third-from-last element is the interesting one.

Once <destination> has been worked out FontMerge merges the fonts from <source> into it. FontMerge automatically overflows into a brand new font directory related to <destination> if necessary, thus removing the need for the user to worry about this.

The auto-overflow works as follows:

If, for example, <destination> is "`ADFS::HardDisc4.$.!boot.Resources.!Fonts.`", then FontMerge will overflow into "`ADFS::HardDisc4.$.!boot.Resources.!Fonts1.`" and, if that gets full, it will overflow into "`ADFS::HardDisc4.$.!boot.Resources.!Fonts2.`" and so on.

Before it starts, FontMerge checks for such overflow directories and start merging into the highest numbered one from the start.

When merging into a directory FontMerge automatically processes font messages files. Messages files for all languages given in the <source> and <destination> are generated.

FontMerge is a directory and should be left a directory. 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, use it is just like using any other command line program.

FontMerge is designed to be run from other desktop applications. It initialises itself as a wimp task to generate wimp error boxes if it has an error, and does Hourglass_Percentage calls as it processes the merge.

The !Boot application structure

The structure of the !Boot application is :

Table 1:

!Boot	Directory
!Run	Obey file
Template	Templates file
!Sprites	Sprites file
!Sprites22	Sprites file
!Help	Text file
Choices	Directory
Boot	Directory
Desktop	Desktop file
PreDesktop	Obey file
PreDesk	Directory
!Configure	Obey file
Tasks	Directory
!Configure	Obey file
Tasks	Directory
Resources	Directory
!Fonts	Directory

Table 1:

!Scrap	Directory
!System	Directory
!ARMovie	Directory
!Configure	Directory
Configure	Directory
2DTools	Sprites file
Monitors	Directory (mode files)
Textures	Directory
RTexture	BASIC file
Utils	Directory
BootRun	Obey file
DeskRun	Obey file
VProtect	Relocatable module
Library	Directory
Repeat	BASIC program
AddApp	Transient utility
AppSize	Transient utility
Do	Transient utility
IfExists	Transient utility
FontMerge	Directory
!Run	Obey file
FontMerge	Absolute image
Messages	Text file

16 File system locking

5

FSLock provides protection against inadvertent (and malicious) changing of the CMOS RAM contents and hard disc contents. To do this is intercepts the SWIs to update CMOS RAM and hard disc contents and returns an error instead. Of course, a machine which allowed no hard disc updates is not very useful, so two areas of the disc have been left unprotected: $.public $.!Boot.Resources.!Scrap.ScrapDir The first is to allow general file storage, the second is to allow Scrap transfers of files between applications.

The changes to Reset behaviour are to close the loophole of delete-power-on and to ensure a reset really always starts the machine afresh.

Technical Background

Old reset behaviour

On RISC OS 3.10 Reset performed the following functions:

Power-On-Self-Test (POST)
Clear RAM
Check keyboard for CMOS RAM clearing Initialise the OS

POST was performed on power-on resets only.

Clearing RAM depended on whether *fx200,2 had been done before the reset, and whether it was a power-on reset or not.

This whole sequence had many variantions 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.

New reset behaviour

Reset caused by pressing the reset button at the back of the case and Reset caused by powering the machine on perform the same function.

This is ensured by the hardware only ever generating the 'power on' flavour of reset. The full sequence of reset operations is performed.

All power-on and reset button resets will:

- Perform power-on self test

- Clear RAM
- Check for CMOS RAM clearing
- Initialise the OS

Reset cause by Break being pressed in combination with one or other of Ctrl or Shift will skip the POST and CMOS RAM clearing, but otherwise function the same. In other words this sort of reset will:

- Clear RAM
- Initialise the OS

Hardware CMOS protection

CMOS RAM clearing is controlled by what keys are held down, and by the state of the CMOS protection connector inside the machine. With the connector in the unprotected position, the CMOS RAM clearing funtions in the same way as it did in RISC OS 3.10. With the protection connector in the protected position the machine's CMOS RAM is protected against being altered in this way.

If the protection connector is in the protected position, CMOS RAM will not be changed in any way as part of any reset sequence (no matter what keys are held down).

The FSLock Module

This module performs the function of protecting the CMOS RAM and hard disc against unwanted modification. It operates in three states:

Fully unlocked

In this state the machine has no password allocated to it. The machine can have its hard discs or configuration modified. This state persists over any sort of reset. A delete-power-on will set the machine to this state.

Partially unlocked

In this state the machine has a password allocated to it. The machine can have its hard discs (1) and configuration modified, although if reset the machine will revert to being locked.

Locked

In this state the machine has a password allocated to it. The machine is protected against having its hard discs (2) or configuration modified. The machine will stay in this state if it is reset.

1. Not all discs on all filing systems are protected. The FSLock module will protect drives 4-7 on any one filing system. The !Configure sets FSLock to protect the ADFS hard discs 4-7 by default.
2. It should be noted that if the whole hard disc was protected against modification that the system would be fairly useless. To get around this FSLock permits writing to anything with the following directories:

```
$.Public
$.!Boot.Resources.!Scrap.ScrapDir
```

Permitted passwords

Max/min size - context sensitive

OS_Byte (&FD)

This reads the last reset type. On this system it will always return 1 - "power-on reset".

SWI calls

In these SWIs the lock status is one of these values:
0 - Fully Unlocked
1 - Partially unlocked
2 - Locked

FSLock_Version (SWI & xxx)

Returns information describing the module

In

-

Out

R0 = version number * 100
R1 = Pointer to module's workspace

Use

This SWI returns information describing the module. R0 gives the module's version number and R1 gives a pointer to the module's workspace.

FSLock_Status (SWI & xxx)

Returns the current lock status and the locked filing system

In

-

Out

R0 = lock status
R1 = filing system locked (undefined if status = 0)

Use

This SWI returns the current lock status and the filing system which has been locked. The filing system number of the locked filing system is given.

FSLock_ChangeStatus (SWI & xxx)

In

R0 = new status
R1 = old password
R2 = new password
R3 = new filing system number

Out

-

Use

This table describes when the passwords must be given:

Old lock status

		0	1	2
New	0	-	O	O
lock	1	N	ON	O
status	2	N	-	ON

O - Old password must be given

N - new password and filing system number must be given

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 existance.

***Commands**

*FSLock_Lock

Locks the computer from the partially unlocked state

Syntax

*FSLock_Lock

Use

This command will lock the machine from the partially unlocked state. If the machine is fully unlocked or locked then a suitable error message will be given.

*FSLock_ChangePassword

Changes the filing system and password

Syntax

FSLock_ChangePassword <FSName> |New password |New password |Old password|||

Use

This command changes the 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 will be given where the password can be typed in without it appearing on screen. Hyphens ('-') will be displayed for each character typed.

*FSLock_Unlock

Fully unlocks the computer

Syntax

FSLock_Unlock |-full| |Password|

Use

This unlocks the machine.

If the -full switch is given then the machine will be fully unlocked, otherwise a partial unlock will be done. If the password is not given then the user will be prompted for the password. At this prompt all characters are displayed as hyphens ('-') so that reading over the shoulder won't work. If the machine is already in the required state (partially or fully unlocked) then an appropriate error will be given.

*FSLock_Status

Displays the machine's status.

Syntax

FSLock_Status

Use

This command displays the machine's status.

17 Miscellaneous items

5

Character Input

OS_Byte 129 (SWI &06)

On exit R1 returns the value of &A5 for RISC OS 3.X when reading the OS version identifier.

18 User interfaces

This chapter brings together technical details of some of the new User interfaces that will be used in RISC OS 3.X.

This information is to enable developers to get an idea of how the changes the operating system have affected the User interface.

This chapter may not be present in future versions of this manual as applications will be described in the User documentation.

Task manager user interface

Task manager and large RAM sizes

The Task manager's Task display window does not cope well in this area at present. The combination of a 16-fold increase in maximum RAM size and an 8-fold decrease in page size means that the existing linear 1 pixel-per-page model will not be practical on large memory machines.

The window now represents the total memory size. It increments in single pages when small, but increases exponentially as you drag the area larger.

Mode Chooser user interface

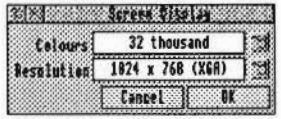
Mode changes are handled by a new application, the Mode Chooser.

The Mode Chooser makes the choice of screen mode easy for ordinary users. The old mode number scheme was complicated and difficult to understand.

With the Mode Chooser, screen modes are described using the number of colours provided and the resolution of the screen. These may be linked with terms such as VGA and SVGA.

Mode Chooser window

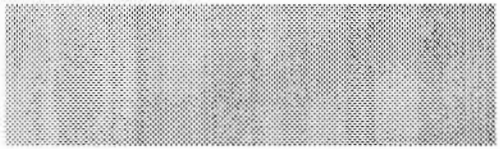
This is the Mode Chooser window:



When this window is opened, the fields show the current screen mode. The colour menu shows the colours available. The resolution menu shows the resolutions possible.

Colours menu

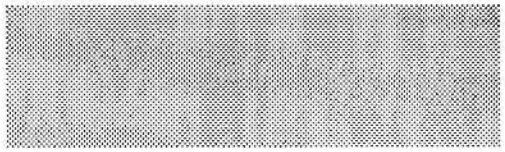
This is the colours menu.



The colours menu is displayed by clicking on the menu icon. The user can then choose on of the colour resolutions.

Resolution menu

The is screen resolution menu:



How it operates

The Mode Chooser uses the new kernel screen mode enumeration facilities to determine what is actually possible with the hardware configuration. It is able to select a high frame rate mode when running with VRAM screen memory and to offer trade-offs when running in a DRAM only system.

None of the colour or resolution entries are greyed out, even though it may not be possible to display that combination of colour and resolution.

When a choice which is not feasible is chosen, the behaviour of the system depends upon which menu the most recent selection was made in. The most recent choice is deemed the most important to the user.

If a **resolution** was chosen the currently selected number of colours is used if possible, otherwise the number of colours are reduced until the resolution chosen becomes possible. 256 colours are reduced to 16 colours. 256 greys are reduced to 16 greys.

If a **colour** setting was chosen: the currently selected resolution is used if possible, otherwise the resolution is reduced until the number of colours chosen becomes possible.

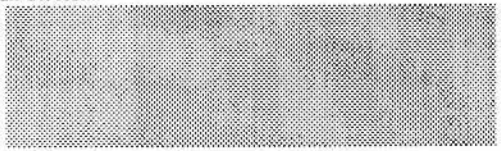
It may not be possible to actually change into the chosen screen mode due to memory constraints. If this occurs the Mode Chooser gives an error message; either, 'There is not enough memory fitted' or 'Not enough free memory - nnnK of memory is required for this screen mode'.

Clicking OK changes to the screen mode defined by the colour and resolution choice or produces an error message explaining why it cannot do so.

Clicking Cancel closes the window, and discards the settings chosen by the user.

Choosing special screen modes using the icon bar menu

The Screen mode option on the Icon bar menu displays a dialogue box which allows nonstandard screen modes to be accessed. It consists of a writable icon and an OK icon.



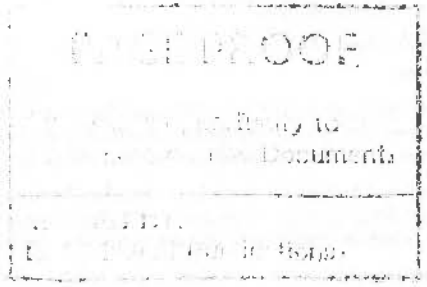
This icon allows screen modes to be chosen by the old method of entering a mode number. It also allows access to screen modes not selectable by the fixed choice of colours and resolutions available in the mode Chooser window. Screen modes can be chosen using the Mode specifier syntax.

Clicking on OK causes the screen mode to be selected, or an error to be generated.

Whenever this dialogue box is opened it is set to the current screen mode (in either mode number or mode selector syntax.

This mode specifier format is also used by several SWI calls and the star command *WimpMode.

This is defined on page XX.



Monitor definition user interface

Screen Modes are selected using the Mode Chooser. However, before selecting your Mode, you need to define the monitor type you are using.

The monitor types available have been expanded, there are now six monitor types and an auto monitor type.

This is a diagram of the modified Screen dialogue window from !Configure.

Monitor types: The following monitor types are available:

- CGA (15kHz)
- Multiscan (15kHz - 35kHz)
- VGA (31kHz)
- SVGA (31kHz - 35kHz)
- XGA (31kHz - 64 kHz)
- XGA (31kHz - 78 kHz)
- Auto

The monitor types are defined as ModeInfo files, these are supplied in the directory !Configure.Monitors with an Auto entry added to the end. The configured monitor is ticked, unless the configured monitor isn't one of the !Configure.Monitors files.

Colours and Resolution: These are selected in the same way as with the mode selector. If monitor Auto is selected then both these fields will come up 'Auto'; you won't be able to change them and the menus won't be available.

Blank delay: This works in the same way as in RISC OS 3.1, except the action is only applied if OK is clicked.

Vertical adjust: This is the same as in RISC OS 3.1, except that it isn't writeable and its effect is only applied if OK is clicked.

Default: This causes default values to be filled in. The default values will match the factory shipped values.

Cancel: A click Select quits the box, no further action. A click Adjust fills in the values in force when the box was last displayed.

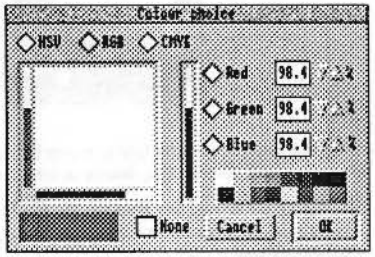
OK: Apply and save the values in the dialogue box.

Colour picker user interface

The user interface for three colour models, RGB, HSV, CMYK is described here.

Common elements of the user interface

This section discusses the workings of the User interface from a technical viewpoint.



Colour model selection area

This is the area at the top of the window containing the radio buttons. These select which colour model is currently active. There can only be one selected colour model – both clicking Select and Adjust selects the desired colour model.

Colour model area

This is the area below the radio buttons controlled by the currently active colour model. There are several different types of colour model can be selected.

The buttons and sliders always remain active even if the None button (no colour) is selected. Adjustments in the colour model area automatically deselect 'None'. The sliders do not become hatched if None is selected; the hatching on the colour patch (see below) is sufficient.

Action button area

This is the area at the bottom of the dialogue box which contains the colour patch, the None option button and the action buttons.

The colour patch is filled with the currently selected colour. If None is selected then the colour patch is cross hatched with black lines on a white background. When the user moves any sliders in the colour model area, or makes any other adjustments the colour patch follows the changes immediately.

The None button is optional. It is controlled by the client application. The button is an option button which toggles on/off with both click Selects and click Adjusts.

The action buttons Cancel and OK are always present. Selecting these causes the dialogue box to do the appropriate action and close. Clicking Adjust keeps the dialogue box open. The OK button is always the default, this can also be selected by pressing the Return key. The Escape key has the same effect as clicking on Cancel.

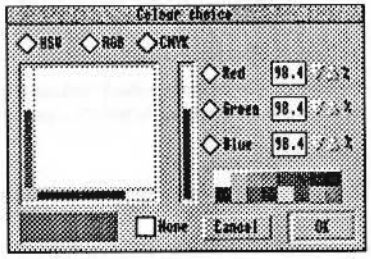
Colour model areas

The standard colour models, RGB, CMYK and HSV are used.

As a general rule, if one of the adjustments is altered, the other things affected by this follow the change immediately. For example, if the central slider is moved in the RGB model, the colour slice changes, the red writeable field changes and the colour patch changes.

RGB model

In this model the user selects a proportion of each of the colours red, green and blue. This is combined to give a colour that the user sees. This is the most natural colour model for a computer displaying on a standard colour monitor.



The three sliders control the red, green and blue amounts added to give the colour. The colour of a slider indicates which component that slider adjusts. The component each slider adjusts is controlled by the radio buttons in the colour model area. The slider closest to the buttons adjusts the component selected by the radio buttons; the other two sliders take up the other two components.

The **large rectangle** to the left shows a slice through the colour cube which corresponds the two sliders next to it. A click anywhere in this slice selects the corresponding position in the RGB colour cube. Adjusting the slider detached from

this slice changes which slice is taken from the cube. In this example the slice has a constant red component and has the green component varying up it and the blue component varying across it.

The **radio buttons** determine which of Red, Green and Blue remains constant across the slice through the RGB colour cube. Only one of these buttons can be selected at a time.

The **writeable fields** allow the user to specify as a percentage (with 1 decimal point) the proportions of Red, Green and Blue. The arrows to the right of these fields allow tweaking of the component. A Click on the arrows will change the value by 0.1%, a Shift-Click by 1%.

The **desktop colour buttons** allow quick selection of one of the desktop colours.

Using the RGB model

Clicking inside the **colour slice through the RGB cube** selects that colour. A click anywhere on the colour slice sliders selects that value of that component.

Clicking on the **slider close to the radio buttons** changes the colour slice to using that as its constant component.

Clicking on the **radio buttons** changes which component is constant on the colour slice. The colour slice is updated to follow the change.

Clicking on the **writeable fields** adjusts the values by 0.1% (or 1% if using shift-click). The sliders follow the change. Accepted values: 0-100 in steps of 0.1.

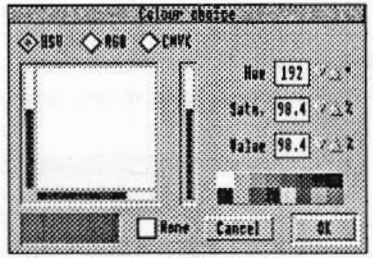
Clicking on the **desktop colour buttons** chooses these colours. The sliders and writeable fields are filled in with the corresponding values.

HSV model

It is assumed in this description that the user is familiar with the HSV colour model.

- Hue gives a colour.
- Saturation gives how far the desired colour is from white to the full colour.
- Value how bright the white is.

This model is more natural for a user to understand.



Referring to the dialogue box it is very like the RGB window. It behaves very similarly. Here are the differences in interpretation.

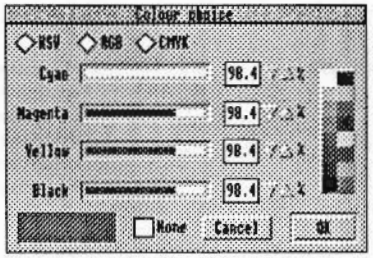
The colour slice is a slice through the HSV cone with the left edge being on the cone's axis, the bottom at the apex of the cone, and the top being a radius of the top of the cone.

This gives a triangle of colour in the top left half of the rectangle where the left edge is from black at the bottom to white at the top, and it fades across to a strong colour along the diagonal bottom left top right edge.

The left slider controls Value, the bottom slider controls Saturation and the right vertical slider controls Hue. The Hue field is an integer from 0-359 degrees which wraps round from 359 back to 0. Saturation and Value are percentages as in the RGB fields.

CMYK model

Here there is no colour slice. This model is most natural for printing where Cyan, Yellow, Magenta and Black inks are used. This is very hard to represent visually, so no attempt to do so has been made. The 4 sliders work as in the RGB model, adjusting the value they correspond to.



Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U