RISC OS
PROGRAMMER'S REFERENCE MANUAL
Volume I

Acorn

Published by Acorn Computers Technical Publications Department

The product described in this manual and products for use with it are subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this manual) are given by Acorn Computers Limited in good faith. However, Acorn Computers Limited cannot accept any liability for any loss or damage arising from the use of any information or particulars in this manual.

This product is not intended for use as a critical component in life support devices or any system in which failure could be expected to result in personal injury.

If you have any comments on this manual, please complete the form at the back of the manual, and send it to the address given there.

Acorn supplies its products through an international dealer network. These outlets are trained in the use and support of Acorn products and are available to help resolve any queries you may have.

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

Published by Acorn Computers Limited
ISBN 1 85250 111 6
Edition 1
Part number 0470,291
Issue 1, October 1991

Contents

About this manual

Summary of contents

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

Part 1

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

Parts 2 to 5

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

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

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

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

Appendices

The Appendices contain:

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

Tables

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

Indexes

The separate volume of Indexes contains:

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

Conventions used

Certain conventions are used in this manual:

Hexadecimal numbers

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

&FFFF (65535)

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

Typefaces

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

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

Command syntax

Special symbols are used when defining the syntax for commands:

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

Programs

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

- BBC BASIC examples omit any line numbering
- BBC BASIC Assembler programs do not show the structure needed to perform the assembly
- ARM Assembler programs assume that header files have been included that define the SWI names as manifests for the SWI numbers. See the chapter entitled *An introduction to* SWIs on page 1-21
- C programs assume that similar headers are included; they also do not show the inclusion of other headers, or the calling of `main()`.

Finding out more

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

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

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

- Desktop C
- Fortran 77
- ISO-Pascal

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

Reader comments

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

Part 1 – Introduction

1 An introduction to RISC OS

Introduction

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

Structure

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

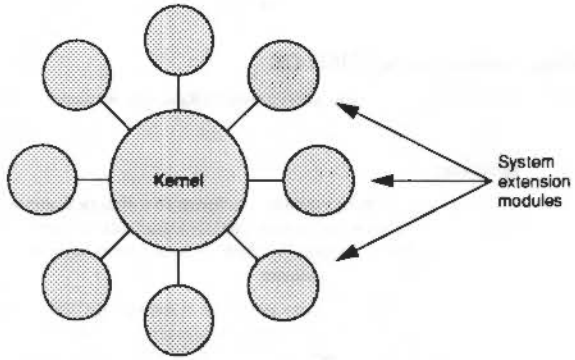


Figure 1.1 The structure of RISC OS

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

Facilities

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

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

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

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

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

Altering and extending RISC OS

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

Modules

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

You can also rewrite any of the standard RISC OS modules. Your replacement must provide the same entry points, and return values in the same way – but its internal workings can be functionally different. See the chapter entitled *Modules* on page 1-191 for further details.

Vectors

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

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

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

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

For more information, see the chapter entitled *Software vectors* on page 1-59.

How RISC OS is written

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

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

To use RISC OS effectively, it helps to have a working knowledge of the ARM processor and of ARM assembler yourself. The chapter entitled ARM *Hardware* on page 1-7 provides a brief introduction to the ARM processor and the set of chips that support it. The appendix entitled *Appendix A: ARM assembler* on page 6-295 will give you a more detailed introduction to the ARM's assembly language.

How RISC OS is supplied

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

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

There is an attendant disadvantage:

- it is harder to upgrade ROMs than a disc.

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

The history of RISC OS

This manual describes RISC OS 3, which was developed from RISC OS 2. This in turn derives from the Arthur operating system, which was the original operating system written for the Archimedes computer.

Two different versions of RISC OS 2 were released:

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

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

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

Arthur

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

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

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

RISC OS 2 documentation

Because many users may prefer not to upgrade from RISC OS 2 to RISC OS 3, we advise you to write applications so that they will still run under both versions. This will maximise your potential market with very little extra effort.

To help you in this, we say explicitly whenever a facility or feature is specific to a version of RISC OS. We've derived this manual directly from the RISC OS *Programmer's Reference Manual* written for RISC OS 2; any other changes or additions you notice have been made to improve clarity and accuracy.

2 ARM Hardware

Introduction

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

ARM chip set

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

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

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

Other components

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

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

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

Schematic

The diagram below gives a schematic of an Archimedes 400 series computer, which may be viewed as typical of a RISC OS computer:

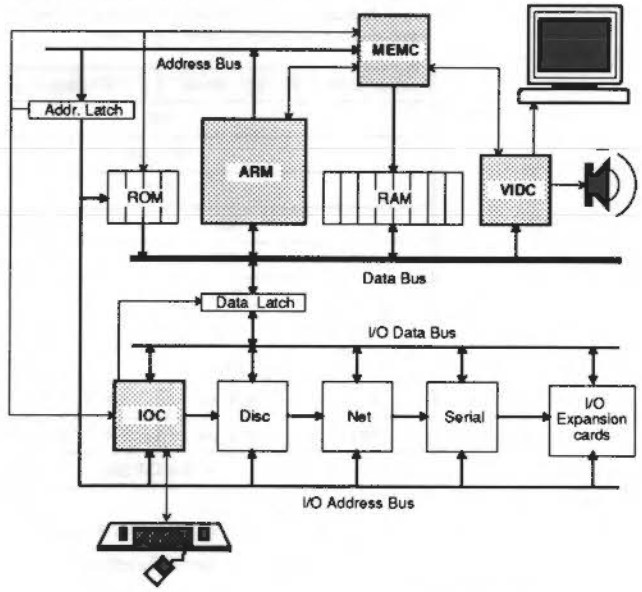


Figure 2.1 Architecture of an Archimedes 400 series computer

The ARM processor

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

RISC and CISC processors

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

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

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

Advantages of RISC

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

- it has a high performance
- it uses much less power than comparable CISC chips
- it is cheaper to produce than CISC processors, making RISC OS computers cheaper for you to buy
- it is much simpler to learn how to program the chip effectively.

ARM 2 and ARM 3

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

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

Word size

The ARM uses 32 bit words. Each instruction fits in a single word. At any one time, the processor is dealing with three instructions:

- one instruction is executed
- the next instruction is simultaneously decoded
- the one after that is fetched from memory.

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

Processor modes

The ARM has four different modes it can operate in:

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

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

Changing mode

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

Registers

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

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

Figure 2.2 ARM registers

Only two of the registers have special functions:

- R15 is used as the program counter (PC) and program status register (PSR)
- R14 (and R14_svc, R14_irq, R14_fiq) are used as subroutine link registers.

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

- R13 (and R13_svc, R13_irq, R13_fiq) are used as private stack pointers for the different processor modes.

All the remaining registers are general purpose.

R15 – program counter and status register

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

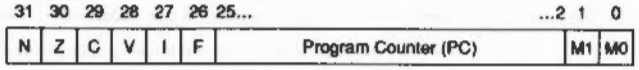


Figure 2.3 Bit usage in R15 by the PC and the PSR

- bits 0 and 1 are the processor mode flags M0 and M1
  - 00 User mode
  - 01 FIQ mode
  - 10 IRQ mode
  - 11 SVC mode
- bits 2 - 25 are the program counter
- bit 26 is the FIQ disable flag F
  - 0 Enable
  - 1 Disable
- bit 27 is the IRQ disable flag I
  - 0 Enable
  - 1 Disable
- bits 28 - 31 are condition flags:
  - V oVerflow flag
  - C Carry flag
  - Z Zero flag
  - N Negative flag

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

R14 – subroutine link registers

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

R13 – private stack pointers

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

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

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

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

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

Instruction set

You will find details of the ARM's Instruction set in the appendix entitled *Appendix A: ARM assembler* on page 6-295.

The VIDC chip

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

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

Buffers

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

- video data
- cursor data
- sound data.

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

Video

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

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

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

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

If needed, support is provided for:

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

Cursor

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

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

Sound

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

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

The IOC chip

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

Internal functions

The following internal functions are provided by the IOC chip:

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

Peripheral control

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

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

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

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

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

The MEMC chip

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

Memory support

MEMC provides the control signals needed by the memory:

- timing and refresh control for dynamic RAM (DRAM)
- control signals for several access times of read-only memory (ROM) – 450ns, 325ns, and 200ns.

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

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

Memory mapping

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

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

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

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

Page size

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

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

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

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

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

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

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

Figure 2.4 Page sizes for Archimedes computers

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

Number of pages programmed

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

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

Protection levels

Three protection levels are provided:

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

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

RISC OS does not use the Operating System mode.

Memory map

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

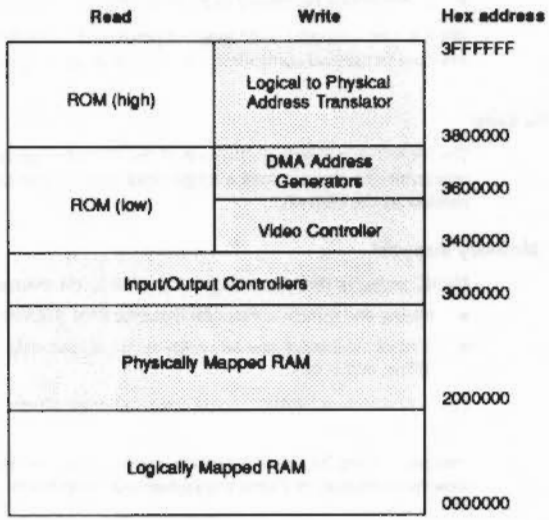


Figure 2.5 Memory map of a typical RISC OS computer

DMA support

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

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

Finding out more

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

- ARM assembler is summarised in the appendix entitled *Appendix A: ARM assembler* on page 6-295
- ARM assembler is thoroughly covered in the manual supplied with the *Desktop Assembler*, available from your Acorn supplier
- The ARM chip set is described in much greater detail in the *Acorn RISC Machine family Data Manual*. VLSI Technology Inc. (1990) Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-781618-9.

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

3 An introduction to SWIs

Introduction

The main way you can access the routines provided by RISC OS is to use a SWI instruction. SWI stands for **S**oft**W**are **I**nterrupt, and is one of the ARM's built-in instructions.

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

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

SWI numbers and names

RISC OS can work out what routine you require because the SWI instruction code contains a 24-bit information field which identifies a routine uniquely. This field is known as the SWI *number*. The section entitled SWI *numbers in detail* on page 1-24 describes how SWI numbers are allocated.

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

Parameters and results

Obviously, you need to be able to pass values to SWI routines (*parameters*), and must be able to read values back (*results*). The ARM registers are used to pass information between the user and RISC OS. In general, you will use R0 to pass the first parameter, and then enough registers after that to pass the rest.

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

Fortunately it is rare that a routine needs to use more than 4 or 5 registers.

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

An example

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

Calling from Assembler

In assembler you could write:

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

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

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

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

Calling from BBC BASIC

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

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

BBC BASIC Assembler

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

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


BBC BASIC

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

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

Calling from C

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

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

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

More about SWI numbers and names

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

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

SWI name prefixes

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

Error handling – an introduction

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

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

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

For further details, see the chapter entitled *Generating and handling errors* on page 1-37.

SWI numbers in detail

The 24 bits used to encode the SWI number in the instruction allow SWIs in the range 0 - &FFFFFF (16777215) to be used. This SWI 'address range' is divided up into several parts under RISC OS. For example, SWIs in the range 0 - &3FFFF (262143) provide the basic operating system functions. (Only a small proportion of these are currently used, however.) Modules can provide their own SWIs, and these must be given unique numbers to avoid clashes.

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

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

Bits 20 - 23

These are used to identify the particular operating system that the SWI expects to be in the machine. All SWIs used under RISC OS have these bits set to zero. Under RISC iX, bit 23 is set to 1 and all other bits are set to zero.

Bits 18 - 19

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

Bit number 19	18	Meaning
0	0	Operating system
0	1	Operating system extension modules
1	0	Third party resident applications
1	1	User applications

Thus OS SWIs, such as OS_WriteC, have both bits clear.

Modules such as filing systems, device drivers for expansion cards, and the Font manager have bit 18 of their SWIs set, so their SWI numbers start at &40000. Note that this can include system extension modules written by third parties.

Any SWIs provided by application software that is distributed by other software houses should have bit 19 set and bit 18 clear.

Bit 17

This is used to determine the action taken on errors. It is the 'X' bit. Error handling in SWIs is described in the chapter entitled *Generating and handling errors* on page 1-37.

Bits 6 - 16

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

Bits 0 - 5

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

000010nnnnnnnnnnn000000 to
000010nnnnnnnnnnn111111

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

Technical details

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

1. The contents of R15 are saved in R14_svc (the SVC mode subroutine link register).
2. The M0 and M1 bits of R15 are set (the processor is forced to SVC mode) and the I bit is also set (IRQ is disabled).
3. The PC bits of R15 are forced to &08.
4. The instruction at &08 is fetched and executed. It is normally a branch to the code that RISC OS uses to decode SWIs.
5. RISC OS uses the PC bits of the return address held in R14_svc to pick up a copy of the SWI instruction.
6. Interrupts are restored to the state they were in when the SWI was issued. This is done by setting the I bit in R15 to the value of the equivalent bit in R14_svc.
7. The V bit of the return address held in R14_svc is cleared, unless the SWI was OS_BreakPt or OS_CallAVector. (This is done for the error handling system – see the chapter entitled *Generating and handling errors* on page 1-37.)
8. RISC OS looks at the 24 bit SWI number field held in the SWI instruction, and uses it to decide where to branch to.
9. If the SWI does not use a vector, RISC OS will branch directly to the actual SWI routine.

   If the SWI does use a vector, RISC OS branches to the routine that calls the vector. Unless you have claimed the vector, this will execute the actual SWI routine.
10. The SWI routine is executed.
11. Any error handling is performed.
12. Any call back handling is performed.
13. Control is returned to your program by using the instruction:

    ```
    MOVS R15, R14_svc.
    ```

    This restores both the mode you were in when you called the SWI, and the interrupt status. Note however that a few SWIs (such as OS_IntOn, which enables interrupts) deliberately alter the mode and/or interrupt status so they are not restored on exit.

    If an error is being returned by setting the V bit, the instruction

    ```
    ORRS R15, R14_svc, #V_bit
    ```

    is used instead.

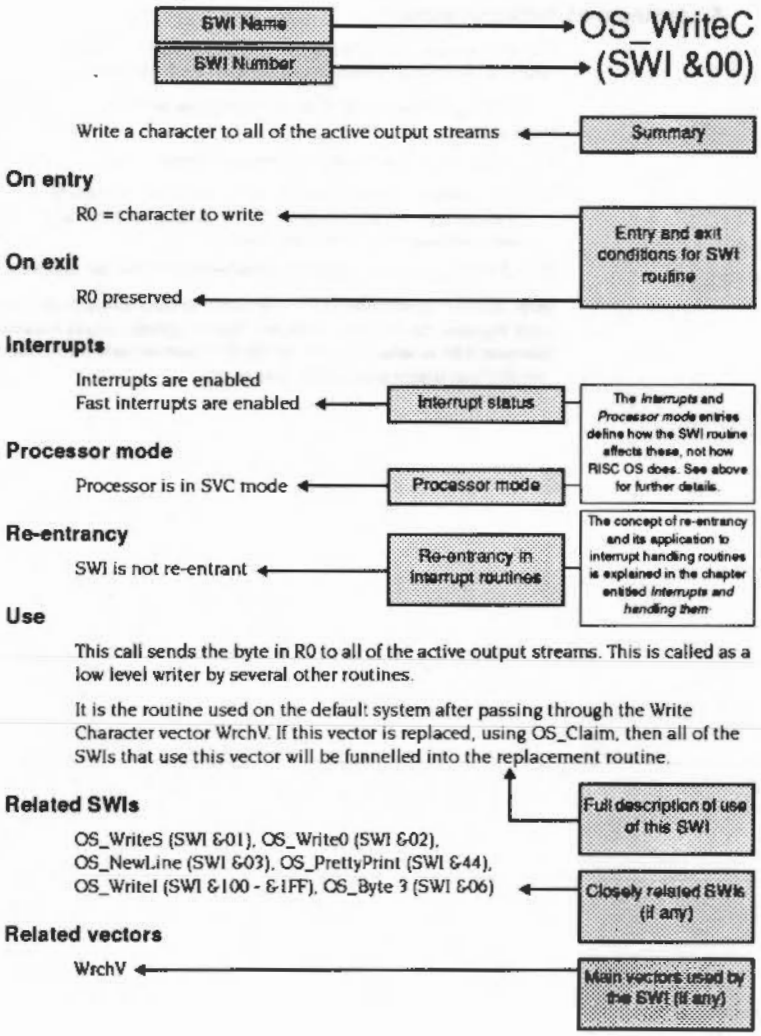
An example of documentation

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

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

- SWIs are decoded and executed as outlined above.
- The V flag is cleared if there is no error, it is set if there is an error, and R0 will then point to an error block. See the chapter entitled *Generating and handling errors* on page 1-37 for further details.
- Other registers and flags are preserved across the call, unless stated otherwise.

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



Important notes

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

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

Calling SWIs from SWI routines

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

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

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

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

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

Error handling with vectored SWIs

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

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

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

4 * Commands and the CLI

Introduction

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

```
*Time
```

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

This chapter introduces you to * Commands and the CLI; the chapter entitled *The CLI* on page 2-429 describes them in more detail.

Command Line mode

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

OS_CLI and the CLI

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

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

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

The * Command version is obviously more convenient.

* Commands v. SWIs

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

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

Their simplicity also leads to some disadvantages:

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

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

Documentation

Each * Command is documented in the relevant chapter. For example, *Time is described in the chapter entitled *Time and Date* on page 1-428. You will find many of the miscellaneous * Commands that the kernel supplies in the chapter entitled *The CLI* on page 2-429. (This chapter also details the OS_CLI SWI.)

An example of documentation

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

Name of * Command → *Time

Displays the day, date and time. ← Summary

Syntax

*Time ← Syntax

Parameters

None ← Parameters

Use

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

Example

*Time ← Example of use

Related commands

None ← Closely related * Commands (if any)

Related SWIs

OS_Word 14,0 (SWI &07), OS_Word 15 (SWI &07), OS_ConvertStandardDateAndTime (SWI &C0), OS_ConvertDateAndTime (SWI &C1) ← Closely related SWIs (if any)

Related vectors

WordV, WrchV ← Main vectors used by the * Command (if any)

Using * Commands

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

Issuing * Commands from applications with command lines

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

When you write applications that provide a command line , they too should recognise any '*' prefixes, and call OS_CLI.

Issuing * Commands from assembler

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

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

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

Issuing * Commands from BASIC

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

Directly from programs

You can issue them directly from your program:

```
*TIME
```

The OSCLI keyword

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

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


Calling OS_CLI directly

Of course, you can also call OS_CLI directly, as outlined in the section entitled *Calling from* BBC BASIC on page 1-22. You can either use the SYS keyword:

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

or more simply:

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

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

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

.TIMESTR        EQUS "TIME"             ; Define the * Command text
                EQUB 0                  ; Terminating null for text
                ALIGN
```

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

Issuing * Commands from C

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

The procedure system ()

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

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

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

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

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

Calling OS_CLI directly

Alternatively, you could directly call OS_CLI:

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

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

Changing and adding * Commands

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

- claiming vectors
- replacing modules
- adding new modules.

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

Using vectors

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

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

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

See the chapter entitled *Software vectors* on page 1-59 for further details of how to use vectors.

Replacing modules

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

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

Adding modules

If you write a new module, it can provide * Commands, in exactly the same way as any of the system modules. See the chapter entitled *Modules* on page 1-191 for details of how to write a module.

5 Generating and handling errors

Introduction

It is reasonable to expect that most SWIs can generate an error. For example, if you pass poor parameters you would expect the SWI routine to tell you about it.

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

Error handling

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

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

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

Types of SWIs

These two types of SWI are known respectively as *error-generating* and *error-returning* SWIs. For every SWI, you can call either version, depending on whether you want to detect the error yourself, or leave the current error handler to deal with it. All the examples in the chapter entitled *" Commands and the* CLI were error-generating SWIs. If you want to call an error-returning SWI, with bit 17 set:

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

Error blocks

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

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

An error block must be word-aligned, and must be no more than 256 bytes long.

Error numbers

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

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

The middle two bytes identify what generated the error. Third parties generating their own errors must apply to Acorn for an identifier. The following error ranges have been reserved:

Range	Error generator
&000 - &0FF	Operating system - BBC-compatible error
&100 - &11F	OS_Module errors
&120 - &13F	OS_ReadVarVal/SetVarVal errors
&140 - &15F	Redirection manager errors
&160 - &17F	OS_EvaluateExpression errors
&180 - &19F	OS_Heap errors
&1A0 - &1AF	OS_Claim/Release errors
&1B0 - &1BF	OS_ChangeEnvironment errors
&1C0 - &1DF	OS_ChangeDynamicArea errors
&1E0 - &1EF	OS_CLI/miscellaneous errors
&200 - &27F	Font manager errors
&280 - &2BF	Wimp errors
&2C0 - &2FF	Date/time conversion errors
&300 - &3FF	Econet errors
&400 - &4FF	FileSwitch errors
&500 - &5BF	Podule errors
&5C0 - &5FF	Printer driver errors
&600 - &63F	General OS errors
&640 - &6FF	International module errors
&700 - &7FF	Sprite errors
&800 - &87F	Debugger errors
&880 - &8FF	BBC I/O Podule errors
&900 - &97F	Shell CLI errors, and miscellaneous others
&980 - &9FF	Draw errors
&A00 - &A3F	ColourTrans errors

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

The top byte contains flags:

- Bit 31, if set, implies that the error was a serious one, usually a hardware exception (eg the program tried to access non-existent memory) or floating point exception, from which it wasn't possible to sensibly return with V set. In such cases different error ranges are used:

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

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

  If bit 31 is set then these bits (24 - 29) are sometimes used as a suberror indicator; for example ADFS uses them to show what kind of disc error has occurred.

Technical details of error-generating SWIs

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

1. First it informs modules of the error using the SWI OS_ServiceCall, with reason code 6 (Error). This is for the module's information only, so that it can tidy up (close files, and so on) before RISC OS handles the error. The module must not try to handle the error.
2. It then calls the error vector (for a detailed description see the chapter entitled *Software vectors* on page 1-59). By default, this calls the current error handler. You may claim this vector, but again this should be for information only – for example, so that your program can tidy up. The call must subsequently be passed on to the error handler; your program must not try to handle the error.

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

Generating errors

In addition to detecting errors, you might want to generate an error which calls the current error handler, so you can find out about a problem. A common example would be if you detect that Esc is pressed. This is usually a sure sign that the user wants to abandon the current operation. The standard response is for you to acknowledge the escape (see the chapter entitled *Character Input* on page 2-337 for details), and generate an 'Escape' error. This is then dealt with by the current error handler.

To generate the error, you should call the SWI OS_GenerateError. On entry, R0 contains a standard error block pointer. The routine never returns. For example, BASIC's error handler will cause the current BASIC program to terminate, returning control to the command mode, or to execute an ON ERROR statement, if one is active.

Writing system extension code

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

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

SWI Calls

OS_GenerateError (SWI &2B)

Generates an error and invokes the error handler

On entry

R0 = pointer to error block

On exit

Doesn't return – OS_GenerateError (SWI &2B) **or**
V flag is set – XOS_GenerateError (SWI &2002B)

Interrupts

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

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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

Here is an example of how OS_GenerateError would be used:

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

Related SWIs

None

Related vectors

ErrorV

* Commands

*Error

Generates errors

Syntax

```
*Error [error_no] text
```

Parameters

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

Use

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

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

Errors are also generated by the system error handler.

Example

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

Related commands

None

Related SWIs

OS_GenerateError (SWI &2B)

Related vectors

ErrorV

6 OS_Byte

Introduction

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

Because the calls are multi-purpose, they tend to appear in more than one chapter of this manual. This chapter documents OS_Byte in general terms, so that when examples of its use are given later on, you will understand the entry and exit conditions better. The next chapter outlines how OS_Word works.

Parameters

OS_Byte takes one, two or three parameters. The first parameter, passed in R0, is the reason code. This indicates which particular action you require OS_Byte to take. It has the range 0 - &FF. Thus when we talk about 'OS_Byte 81', this is shorthand for 'OS_Byte with R0 set to 81 on entry'. A complete list of the OS_Byte numbers may be found in the *Index of OS_Bytes*.

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

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

Some OS_Byte calls return values. On earlier Acorn computers these were always byte-wide, but on RISC OS computers some of these values may now be too large to fit in a single byte, and should be treated as whole words. For example, if you

were reading the number of spaces left in a buffer using OS_Byte 128, you might read the two 'byte' result returned in R1 (low byte) and R2 (high 'byte' – in fact 24 bits) like this:

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

Calling OS_Byte

You call the OS_Byte SWI in exactly the same way as any other SWI. See the chapter entitled *An introduction to* SWIs on page 1-21 for details.

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

Calls where R0 is less than 128

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

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

Calls where R0 is between 128 and 165 (inclusive)

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

Calls where R0 is between 166 and 255 (inclusive)

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

New Value = (Old Value AND R2) EOR R1

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

Reading and writing values

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

R2 = &FF and R1 = &00

and where the value is set to a particular number:

R2 = &00 and R1 = new value

Altering selected bits

These are the only cases which are stated in the descriptions of OS_Bytes in this guide. Other values of R1 and R2 may be used to alter only selected bits of the status variable. You should:

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

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

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

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

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

Which call to use when

Many of the calls in this last group access the same status variable as the low-numbered calls, between 0 and 127. However, as noted above, the lower group may also perform some other action in addition to changing the variable value. This means that the lower group should be used to alter a variable, whereas the upper group may be used for reading the current value without changing it.

OS_Byte and interrupts

Like most important SWIs, OS_Byte is vectored so you can alter how it works. Before its vector is called, interrupts are disabled. Most OS_Byte routines are so short that there is no need for them to re-enable interrupts – instead they rely on

RISC OS doing this when control is returned to you. Because these OS_Byte routines do not re-enable interrupts they are also used by interrupt handling routines

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

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

Adding OS_Byte calls

You can add your own OS_Byte calls to RISC OS by installing a routine on the software vector that OS_Byte calls use. For full details, see the chapter entitled *Software vectors* on page 1-59.

There is an alternative, but less preferable way of adding OS_Byte calls. If you issue an OS_Byte with a number that RISC OS doesn't recognise, it issues an Unknown OS_Byte *service call* to all modules. Your module can then trap this service call and implement the new OS_Byte. For full details, see the chapter entitled *Modules* on page 1-191.

The *FX command

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

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

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

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

has the same effect as:

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


Calling *FX

The *FX command does not display any returned values; you cannot use it to read the values of status variables from the command line. It is called in the same way as any other * Command; see the chapter entitled * *Commands and the* CLI on page 1-31 for details.

SWI calls

OS_Byte (SWI &06)

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

On entry

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

On exit

R0 preserved
R1, R2 – as returned by individual OS_Byte

Interrupts

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

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant for some values of R0

Use

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

- If R0 is less than 128, then generally only R1 is used to pass further information. These calls set a status variable, and may also perform some other task. R2 is corrupted unless stated otherwise.
- If R0 is between 128 and 165 (inclusive), both R1 and R2 are used to hold parameters, and both registers may contain information on exit from the call. The calls are often used for the results they return.

- For calls with R0 between 166 and 255 (inclusive) on entry, the action is always the same. R0 acts as an index to a status variable, which is altered using the contents of R1 and R2:

  New Value = (Old Value AND R2) EOR R1

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

Related SWIs

OS_Word (SWI &07)

Related vectors

ByteV

* Commands

*FX

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

Syntax

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

Parameters

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

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

Use

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

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

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

Examples

`*FX 218,0,&FF`	*r2 is specified in hexadecimal*
`*FX 247 4_01`	*r1 is specified in base 4*

Related commands

None

Related SWIs

OS_Byte (SWI &06)

Related vectors

ByteV

7 OS_Word

Introduction

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

You can add new OS_Word calls by installing a routine on the software vector that OS_Word uses – see the chapter entitled *Software vectors* on page 1-59. Alternatively you can use the Unknown OS_Word service call, although this is not such a good way to do so, and is hence deprecated – see the chapter entitled *Modules* on page 1-191.

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

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

Parameters

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

Like OS_Byte, OS_Word is multi-purpose, and covers such areas as reading the time and date, setting the screen's 'palette', and reading the definition of a re-definable character.

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

Calling OS_Word

You call the OS_Word SWI in exactly the same way as any other SWI. For details see the earlier chapter entitled *An introduction to* SWIs on page 1-21.

OS_Word and * Commands

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

SWI calls

OS_Word (SWI &07)

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

On entry

R0 = reason code
R1 = pointer to parameter block

On exit

R0 preserved

Interrupts

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

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The action taken by this SWI depends on the reason code passed in R0. In general, OS_Word is used to either read or write a large number of status variables at once. R1 points to a parameter block, the length of which varies depending on the reason code. You should see the individual documentation of each OS_Word for full details.

Related SWIs

OS_Byte (SWI &06)

Related vectors

WordV

8 Software vectors

Introduction

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

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

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

Claiming vectors

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

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

Your own routine can do one of the following:

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

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

An example

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

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

Vector chains

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

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

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

When not to intercept a vector

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

Multiply installing the same routine

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

SWI Calls

OS_Claim (SWI &1F)

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

On entry

R0 = vector number (see page 1-70)
R1 = address of claiming routine that is to be added to vector
R2 = value to be passed in R12 when the routine is called

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered as it disables IRQ

Use

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

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

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

Example

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

Related SWIs

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

Related vectors

All

OS_Release (SWI &20)

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

On entry

R0 = vector number (see page 1-70)
R1 = address of routine that is to be released from vector
R2 = value given in R2 when claimed

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered as it disables IRQ

Use

This call removes the routine, which is identified by both its address and workspace pointer, from the list for the specified vector. The routine will no longer be called. If more than one copy of the routine is claiming the vector, only the first one to be called is removed.

Example

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

Related SWIs

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

Related vectors

All

OS_CallAVector (SWI &34)

Calls a vector directly

On entry

R0 - R8 = vector routine parameters
R9 = vector number (see page 1-70)
V and C flags in R15 = flags to pass to vector

On exit

Dependent on vector called

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

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

Use

OS_CallAVector calls the vector number given in R9. R0 - R8 are parameters to the vectored routine; see the descriptions below for details.

This is used for calling vectored routines which don't have any other entry point, such as some calls to RemV or CnpV. It is also used by system extensions such as the Draw, ColourTrans and Econet modules to call their corresponding vectors.

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

Related SWIs

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

Related vectors

All

OS_AddToVector (SWI &47)

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

On entry

R0 = vector number (see page 1-70)
R1 = address of claiming routine
R2 = value to be passed in R12 when the routine is called

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered as it disables IRQ

Use

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

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

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

Related SWIs

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

Related vectors

All

Use of registers

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

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

Processor modes

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

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

SVC mode

Note that if you call a SWI from a routine that is in SVC mode, you will corrupt the return address held in R14. Consequently, your routine should use the full, descending stack addressed by R13 to save R14 first. See the section entitled *Important notes* on page 1-29 for a more complete explanation of this.

IRQ mode

If your routine will be entered in IRQ mode there are other restrictions. These are detailed in full in the section entitled *Restrictions* on page 1-118.

Returning errors

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

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

Returning from a vectored routine

You should use one of two methods to return from a vectored routine. These are described immediately below; for an example, see the section entitled *An example program* on page 1-98.

Passing on the call

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

```
MOVS PC,R14
```

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

Intercepting the call

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

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

Control will return to the caller of the vector.

List of software vectors

The software vectors are listed below. The following section entitled *Summary of vectors* gives a summary of each vector, and tells you where to find out more about it. A few vectors also merit a more detailed description in the section entitled *Vector descriptions* on page 1-77. Such vectors are indicated in the list below by a dagger '†'. Also, the names of the routines which can cause each vector to be called are in brackets.:

Vector		No	Description
UserV		(&00)	User vector (reserved)
ErrorV		(&01)	Error vector (OS_GenerateError)
IrqV	†	(&02)	Interrupt vector
WrchV		(&03)	Write character vector (OS_WriteC)
RdchV		(&04)	Read character vector (OS_ReadC)
CLIV		(&05)	Command line interpreter vector (OS_CLI)
ByteV		(&06)	OS_Byte indirection vector (OS_Byte)

WordV		(&07)	OS_Word indirection vector (OS_Word)
FileV		(&08)	File read/write vector (OS_File)
ArgsV		(&09)	File arguments read/write vector (OS_Args)
BGetV		(&0A)	File byte read vector (OS_BGet)
BPutV		(&0B)	File byte put vector (OS_BPut)
GBPBV		(&0C)	File byte block get/put vector (OS_GBPB)
FindV		(&0D)	File open vector (OS_Find)
ReadLineV		(&0E)	Read a line of text vector (OS_ReadLine)
FSCV		(&0F)	Filing system control vector (OS_FSControl)
EventV		(&10)	Event vector (OS_GenerateEvent)
InsV	†	(&14)	Buffer insert vector (OS_Byte)
RemV	†	(&15)	Buffer remove vector (OS_Byte)
CnpV	†	(&16)	Count/Purge Buffer vector (OS_Byte)
UKVDU23V	†	(&17)	Unknown VDU23 vector (OS_WriteC)
UKSWIV	†	(&18)	Unknown SWI vector (SWI)
UKPLOTV	†	(&19)	Unknown VDU25 vector (OS_WriteC)
MouseV		(&1A)	Mouse vector (OS_Mouse)
VDUXV	†	(&1B)	VDU vector (OS_WriteC)
TickerV	†	(&1C)	100Hz pacemaker vector
UpcallV		(&1D)	Warning vector (OS_UpCall)
ChangeEnvironmentV		(&1E)	Environment change vector (OS_ChangeEnvironment)
SpriteV		(&1F)	OS_SpriteOp indirection vector (OS_SpriteOp)
DrawV	†	(&20)	Draw SWI vector (Draw_...)
EconetV	†	(&21)	Econet activity vector (Econet_...)
ColourV	†	(&22)	ColourTrans SWI vector (ColourTrans_...)
PaletteV	†	(&23)	Read/write palette vector
SerialV		(&24)	OS_SerialOp indirection vector

All other vectors are reserved by Acorn.

Summary of vectors

Brief details of these vectors are given below.

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

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

About the filing system vectors

Note that the filing system vectors FileV (Vector &08) to FindV (Vector &0D) have 'no default action', ie they return immediately. However, the FileSwitch module (described in the chapter of the same name, starting on page 3-9) OS_Claims the vectors whenever the machine is reset, so effectively the default action is to perform the appropriate filing system routine.

Other vectors and resets

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

UserV

UserV is a reserved vector, and you must not use it. Its default action is to do nothing.

ErrorV

ErrorV is used to indirect all errors from error-generating SWIs and from OS_GenerateError – see page 1-41 for full details. The default action is to call the error handler.

See also the rest of the chapter entitled *Generating and handling errors*; and the chapter entitled *Program Environment* on page 1-277 for more about handlers.

IrqV

IrqV is called when an unknown IRQ is detected. It enables you to add interrupt generating devices of your own to the computer. The default action is to disable the interrupting device. See page 1-78 later in this chapter for full details.

See also the chapter entitled *Interrupts and handling them* on page 1-109, and the chapter entitled *Program Environment* on page 1-277 for more about handlers.

WrchV

WrchV is used to indirect all calls to OS_WriteC – see page 2-13 for full details. The default action is to call the ROM write character routine.

RdchV

RdchV is used to indirect all calls to OS_ReadC – see page 2-357 for full details. The default action is to call the ROM read character routine.

CLIV

CLIV is used to indirect all calls to OS_CLI – see page 2-435 for full details. The default action is to call the ROM command line interpreter.

ByteV

ByteV is used to indirect all calls to OS_Byte – see page 1-50 for full details. The default action is to call the ROM OS_Byte routine.

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

WordV

WordV is used to indirect all calls to OS_Word – see page 1-57 for full details. The default action is to call the ROM OS_Word routine.

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

FileV

FileV is used to indirect all calls to OS_File – see page 3-27 for full details. The default action is to call the ROM OS_File routine (see the note above).

ArgsV

ArgsV is used to indirect all calls to OS_Args – see page 3-42 for full details. The default action is to call the ROM OS_Args routine (see the note above).

BGetV

BGetV is used to indirect all calls to OS_BGet – see page 3-56 for full details. The default action is to call the ROM OS_BGet routine (see the note above).

BPutV

BPutV is used to indirect all calls to OS_BPut – see page 3-58 for full details. The default action is to call the ROM OS_BPut routine (see the note above).

GBPBV

GBPBV is used to indirect all calls to OS_GBPB – see page 3-59 for full details. The default action is to call the ROM OS_GBPB routine (see the note above).

FindV

FindV is used to indirect all calls to OS_Find – see page 3-68 for full details. The default action is to call the ROM OS_Find routine (see the note above).

ReadLineV

ReadLineV is used to indirect all calls to OS_ReadLine – see page 2-417 for full details. The default action is to call the ROM OS_ReadLine routine.

FSCV

FSCV is used to indirect calls to OS_FSControl – see page 3-73 for full details. The default action is to call the ROM OS_FSControl routine.

EventV

EventV is used to indirect all calls to OS_GenerateEvent – see page 1-144 for full details. The default action is to call the event handler.

See also the rest of the chapter entitled *Events*; and the chapter entitled *Program Environment* on page 1-277 for more about handlers.

InsV

InsV is called to place a byte in a buffer. See page 1-80 later in this chapter for full details.

See also the chapter entitled *Buffers* on page 1-153.

RemV

RemV is called to remove a byte from a buffer. See page 1-82 later in this chapter for full details.

See also the chapter entitled *Buffers* on page 1-153.

CnpV

CnpV is called to count the number of entries in a buffer, or to purge the contents of a buffer. See page 1-84 later in this chapter for full details.

See also the chapter entitled *Buffers* on page 1-153.

UKVDU23V

UKVDU23V is called when a VDU 23,*n* command is issued with an unknown value of *n*. The default action is to do nothing – unknown VDU 23s are usually ignored. See page 1-86 later in this chapter for full details.

UKSWIV

UKSWIV is called when a SWI is issued with an unknown SWI number. The default action is to call the unknown SWI handler, which by default generates a `No such SWI` error. See page 1-87 later in this chapter for full details.

See also the chapter entitled *An introduction to SWIs* on page 1-21; and the chapter entitled *Program Environment* on page 1-277 for more about handlers.

UKPLOTV

UKPLOTV is called when a VDU 25,*n* (Plot) command is issued with an unknown value of *n*. The default action is to do nothing – unknown VDU 25s (Plots) are usually ignored. See page 1-88 later in this chapter for full details.

MouseV

MouseV is used to indirect all calls to OS_Mouse – see page 2-207 for full details. The default action is to call the ROM OS_Mouse routine.

VDUXV

VDUXV is called when VDU output has been redirected by setting bit 5 of the OS_WriteC destination flag. This vector is normally claimed by the Font Manager, to implement the Font system (see the chapter entitled *The Font Manager* on page 5-1). If the Font module is disabled, the default action is to do nothing – no output is sent to the VDU. See page 1-89 later in this chapter for full details.

See also the chapter entitled *Character Output* on page 2-1, and the chapter entitled VDU Drivers on page 2-39.

TickerV

TickerV is called every centi-second. It must never be intercepted. See page 1-90 later in this chapter for full details.

UpCallV

UpCallV is used to indirect all calls to OS_UpCall – see the chapter entitled *Communications within* RISC OS on page 1-167 for full details. The default action is to call the UpCall handler.

ChangeEnvironmentV

ChangeEnvironmentV is used to indirect all calls to OS_ChangeEnvironment – see page 1-307 for full details. The default action is to call the ROM OS_ChangeEnvironment routine.

SpriteV

SpriteV is used to indirect all calls to OS_SpriteOp – see page 2-262 for full details. The default action is to call the relevant ROM OS_SpriteOp routine. (In fact there are two claimants for this vector: one intercepts those calls handled by the kernel's sprite routines, the another intercepts those handled by the SpriteExtend module.)

DrawV

DrawV is used to indirect **all** SWI calls made to the Draw module. The default action is to call the ROM routine in the Draw module that decodes and executes SWIs. See page 1-91 later in this chapter for full details.

See also the chapter entitled *Draw module* on page 5-111.

EconetV

EconetV is called whenever there is activity on the Econet. The default action is to display the Hourglass on the screen. See page 1-92 later in this chapter for full details.

See also the chapter entitled *Econet* on page 6-1, the chapter entitled *Hourglass* on page 6-73, and the chapter entitled *NetStatus* on page 6-83.

ColourV

ColourV is used to indirect **all** SWI calls made to the ColourTrans module. The default action is to call the routine in the ColourTrans module that decodes and executes SWIs. See page 1-94 later in this chapter for full details.

See also the chapter entitled *ColourTrans* on page 4-381.

PaletteV

PaletteV is called whenever a call is made to read or write the palette. The default action is to call the ROM routine to read or write the palette. See page 1-96 later in this chapter for full details.

This vector has no default owner under RISC OS 2.

SerialV

SpriteV is used to indirect all calls to OS_SerialOp – see page 3-440 for full details. The default action is to call the ROM OS_SerialOp routine.

This vector has no default owner under RISC OS 2.

Vector descriptions

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

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

IrqV (Vector &02)

Called when an unknown IRQ is detected

On entry

No parameters passed in registers

On exit

—

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in IRQ mode

Use

This vector is called when an unknown IRQ is detected.

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

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

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

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

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

See the chapter entitled *Interrupts and handling them* on page 1-109 for details of how to add interrupt generating devices to your computer, and the chapter entitled *Program Environment* on page 1-277 for more about handlers.

Related SWIs

None

InsV (Vector &14)

Called to place a byte in a buffer

On entry

R0 = byte to be inserted (if byte operation)
R1 = buffer number (bits 0 - 30) and byte/block operation flag (bit 31):
bit 31 clear ⇒ byte operation – R0 holds byte
bit 31 set ⇒ block operation – R2 points to block of length R3
R2 = pointer to first byte of data to be inserted (if block operation)
R3 = number of bytes to insert (if block operation)

On exit

R0, R1 preserved
R2 corrupted – if byte operation;
else R2 = pointer to remaining data to be inserted – if block operation
R3 = number of bytes still to be inserted – if block operation
C = 1 implies insertion failed

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Use

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

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

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

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

See also the chapter entitled *Buffers* on page 1-153 and the chapter entitled *The Buffer Manager* on page 5-407.

Related SWIs

OS_Byte 138 and 153 (SWI &06)

RemV (Vector &15)

Called to remove a byte from a buffer

On entry

R1 = buffer number (bits 0 - 30) and byte/block operation flag (bit 31):
bit 31 clear ⇒ byte operation – R0 holds byte
bit 31 set ⇒ block operation – R2 points to block of length R3
R2 = pointer to block to be filled (if block operation)
R3 = number of bytes to place into block (if block operation)
V flag = 1 if buffer to be examined only, or 0 if data should actually be removed

On exit

R0 = next byte to be removed (examine option) – if for byte operation;
else preserved
R1 preserved
R2 = byte removed (remove option), or preserved – if for byte operation;
else R2 = pointer to updated buffer position – if block operation
R3 = number of bytes still to be removed – if block operation
C = 1 if buffer was empty on entry

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Use

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

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

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

On exit C is used to indicate if the calls actually worked and if a byte or the requested block of data could be obtained.

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

See also the chapter entitled *Buffers* on page 1-153 and the chapter entitled *The Buffer Manager* on page 5-407.

Related SWIs

OS_Byte 145 and 152 (SWI &06)

CnpV (Vector &16)

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

On entry

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

On exit

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

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Use

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

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

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

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

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

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

This call also copes with buffer manager buffers.

See also the chapter entitled *Buffers* on page 1-153 and the chapter entitled *The Buffer Manager* on page 5-407.

Related SWIs

OS_Byte 15, 21 and 128 (SWI &06)

UKVDU23V (Vector &17)

Called when an unrecognised VDU 23 command is issued.

On entry

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

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

This vector is called when a VDU 23,*n* command is issued with an unknown value of *n*, ie it is in the range 18 - 25 or 28 - 31.

The nine parameters sent after the VDU 23 command are stored in the VDU queue. R1 points to the byte holding *n*, and R0 also contains *n*.

The default action is to do nothing – unknown VDU 23s are ignored.

Related SWIs

OS_WriteC (SWI &00)

UKSWIV (Vector &18)

Called when an unknown SWI instruction is issued

On entry

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

On exit

Generates an error by default

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

This vector is called when a SWI is issued with an unknown SWI number. Before this vector is called, the OS tries to pass the call to any modules which have SWI table entries in their header.

The default action is to call the Unused SWI handler, which by default returns a 'No such SWI' error. See the section entitled *Unused* SWI on page 1-285 for full details.

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

See also the chapter entitled *An introduction to* SWIs on page 1-21; and the chapter entitled *Program Environment* on page 1-277 for more about handlers.

Related SWIs

OS_UnusedSWI (SWI &19)

UKPLOTV (Vector &19)

Called when an unknown PLOT command is issued

On entry

R0 = PLOT number

On exit

R0 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

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

By using OS_ReadVduVariables you can read the co-ordinates of the last three points that have been visited, and the one specified in the unknown PLOT command. These are held in the VDU variables 140 - 147. See the entry for OS_ReadVduVariables for full details.

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

The default action is to do nothing – unknown VDU 25s (Plots) are ignored.

Related SWIs

OS_WriteC (SWI &00), OS_Plot (SWI &45)

VDUXV (Vector &1B)

Called when VDU output has been redirected

On entry

R0 = byte sent to the VDU

On exit

R0 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

This vector is called when VDU output has been redirected by setting bit 5 of the OS_WriteC destination flag. When this bit is set, all characters sent to the VDU driver are routed through this vector instead. Note that this only affects the display driver: other output streams such as the printer and *SPOOL file are called as usual, even when VDUXV is used for screen updating.

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

This vector is normally claimed by the Font Manager, to implement the Font system (see the chapter entitled *The Font Manager* on page 5-1). If the Font module is disabled, the default action is to do nothing – no output is sent to the VDU.

See also the chapter entitled *Character Output* on page 2-1, and the chapter entitled VDU Drivers on page 2-39.

Related SWIs

OS_WriteC (SWI &00)

TickerV (Vector &1C)

Called every centi-second

On entry

No parameters passed in registers

On exit

—

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Use

This vector is called every centi-second. It must never be intercepted, as this would prevent other users from being called.

Routines that take a long time (say > 100µs) may re-enable IRQ so long as they disable it again before passing the call on. If you do so, other calls may be made to TickerV in the meantime. Your routine needs to prevent or cope with re-entrancy. One way of ensuring that the code is single threaded is:

- to use a flag in its workspace to note that it is currently threaded, and:
- to keep a count of how many calls to TickerV have been missed while it was threaded, so the count can be examined on exit and corrected for.

Related SWIs

None

DrawV (Vector &20)

Used to indirect all SWI calls made to the Draw module

On entry

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

On exit

Dependent on SWI issued

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

This vector is used to indirect **all** SWI calls made to the Draw module. The default action is to call the ROM routine in the Draw module that decodes and executes SWIs.

The index held in R8 is decoded as follows:

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

See also the chapter entitled *Draw module* on page 5-111.

Related SWIs

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

EconetV (Vector &21)

Called whenever there is activity on the Econet

On entry

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

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

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

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

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

&10	NetFS_StartLoad	R1 = total size of data
&11	NetFS_PartLoad	R1 = amount of data transferred
&12	NetFS_FinishLoad	
&20	NetFS_StartSave	R1 = total size of data
&21	NetFS_PartSave	R1 = amount of data transferred
&22	NetFS_FinishSave	
&30	NetFS_StartCreate	R1 = total size of data
&31	NetFS_PartCreate	R1 = amount of data transferred
&32	NetFS_FinishCreate	
&40	NetFS_StartGetBytes	R1 = total size of data

&41	NetFS_PartGetBytes	R1 = amount of data transferred
&42	NetFS_FinishGetBytes	
&50	NetFS_StartPutBytes	R1 = total size of data
&51	NetFS_PartPutBytes	R1 = amount of data transferred
&52	NetFS_FinishPutBytes	
&60	NetFS_StartWait	start of a Broadcast_Wait
&62	NetFS_FinishWait	end of a Broadcast_Wait
&70	NetFS_StartBroadcastLoad	R1 = total size of data
&71	NetFS_PartBroadcastLoad	R1 = amount of data transferred
&72	NetFS_FinishBroadcastLoad	
&80	NetFS_StartBroadcastSave	R1 = total size of data
&81	NetFS_PartBroadcastSave	R1 = amount of data transferred
&82	NetFS_FinishBroadcastSave	
&C0	Econet_StartTransmission	start to wait for a transmission to end
&C2	Econet_FinishTransmission	DoTransmit returns
&D0	Econet_StartReception	start to wait for a reception to end
&D2	Econet_FinishReception	WaitForReception returns

This vector is normally claimed by the NetStatus module, which uses the Hourglass module to display an hourglass while the Econet is busy. It passes on the call. If the Hourglass module is disabled, the default action is to do nothing. See the chapter entitled *Hourglass* on page 6-73, and the chapter entitled *NetStatus* on page 6-83.

See also the chapter entitled NetFS on page 3-323, the chapter entitled *NetPrint* on page 3-367, and the chapter entitled *Econet* on page 6-1.

Related SWIs

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

ColourV (Vector &22)

Used to indirect all SWI calls made to the ColourTrans module

On entry

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

On exit

Dependent on SWI issued

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Use

This vector is used to indirect **all** SWI calls made to the ColourTrans module. The default action is to call the routine in the ColourTrans module that decodes and executes SWIs.

The index held in R8 is decoded as follows:

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

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

See also the chapter entitled *ColourTrans* on page 4-381.

Related SWIs

ColourTrans_... (SWIs &40740 - &4077F)

PaletteV (Vector &23)

Called whenever the palette is to be read or written

On entry

R0 = logical colour
R1 = type of colour (16,17,18,24,25)
If R4 = 2 (ie writing):
    R2 = 1st flash colour (&BBGGRRxx) – device colour
    R3 = 2nd flash colour (&BBGGRRxx) – device colour
R4 = 1 ⇒ reading palette, 2 ⇒ writing to palette

On exit

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

Use

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

- VDU 19 page 2-79
- OS_Word 12 page 2-187
- OS_ReadPalette page 2-209
- ColourTrans_ReadPalette page 4-431
- ColourTrans_WritePalette page 4-433

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

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

This vector has no default owner under RISC OS 2. However, you can write software that calls this vector – and that works correctly under all versions of RISC OS – by checking the value of R4 on exit to see if the operation is complete. If it is not

complete, you then need to use your own code to read or write the palette. For more information and example code fragments, see the section entitled *Application notes* on page 1-101.

More complex uses of vectors

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

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

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

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

An example program

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

The program claims WrchV, adding a routine that:

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

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


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

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

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

.done_preprocess%

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

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

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

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

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

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

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

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

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

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

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


Application notes

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

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

Reading a palette

The first piece of code reads the palette:

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

readpalette Entry "R4,R9"

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

        TEQ R4,#0
        EXIT EQ

        SWI XOS_ReadPalette     ; returns &B0G0R0xx

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

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

        EXITS VC
        EXIT
        LTORG
```

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

Writing a palette

The first piece of code writes the palette:

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

setpalette "R4,R9"

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

        TEQ R4,#0
        EXITS EQ

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

        EXITS VC
        EXIT
```

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

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

9 Hardware vectors

Introduction

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

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

Reset vector

When the computer is reset, amongst other things:

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

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

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

Hardware exception vectors

The middle group of vectors, except SWI, are under the control of various 'environment' handlers. When the exception occurs, before any of these vectors is called, the ARM processor saves the current program counter (R15) to R14_svc. The ARM is then forced to SVC mode, and interrupts are disabled.

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

These vectors may be set and read as described in the chapter entitled *Program Environment* on page 1-277. Very few programs need to take account of them.

Undefined instruction vector

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

Prefetch abort vector

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

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

Data abort vector

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

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

Address exception vector

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

SWI vector

The SWI vector is called when a SWI instruction is issued. It contains a branch to the RISC OS code which decodes the SWI number and branches to the appropriate location. Before calling this vector, the ARM processor saves the current program counter (R15) to R14_svc. The ARM is then forced to SVC mode, and interrupts are disabled.

You are strongly recommended not to replace this vector.

For full details, see the earlier chapter entitled *An introduction to SWIs* on page 1-21.

IRQ vector

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

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

For full details, see the chapter entitled *Interrupts and handling them* on page 1-109.

FIQ vector

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

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

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

Claiming hardware vectors

If you are the current application, you can change the effect of most of the hardware vectors by installing the appropriate handler. If you are not, then you will have to 'claim' the vector yourself. This is most likely to occur if you are a system

extension module. There is no SWI to claim a hardware vector; instead you have to overwrite it with a `B myHandler` or a `LDR PC, [PC, #myHandlerOffset]` instruction, and do all the 'housekeeping' yourself.

Passing on calls to hardware vectors

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

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

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

Releasing hardware vectors

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

Vector priorities

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

Reset
Address exception
Data abort ↑ High priority
FIQ
IRQ

Prefetch abort ↓ Low priority
Undefined instruction
SWI

10 Interrupts and handling them

Introduction

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

RISC OS deals with the interrupt by temporarily halting its current task, and entering an *interrupt routine*. This routine deals with the interrupting device very quickly – so quickly, in fact, that you will never realise that your program has been interrupted.

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

Devices handled

Amongst the devices which are handled under interrupts on RISC OS computers are the:

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

Expansion cards

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

Device numbers

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

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

The device numbers correspond directly to bits of the interrupt registers held in the IOC chip:

Device number	Corresponds to:
0	Bit 0 of IRQ A registers
1	Bit1 of IRQ A registers
...	
7	Bit 7 of IRQ A registers
8	Bit 0 of IRQ B registers
9	Bit 1 of IRQ B registers
...	
15	Bit7 of IRQ B registers

See the section entitled IOC *registers* on page 1-133 for more details.

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

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

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

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

0	Printer interrupt from 82C710/711
1	Low battery warning
2	Floppy disc Index
3	VSync Pulse
4	Power on reset – this should never appear in normal use
5	IOC Timer 0
6	IOC Timer 1
7	FIQ Downgrade – reserved for the use of the current owner of FIQ
8	Expansion card FIQ Downgrade – this should normally be masked off
9	Sound system buffer change
10	Serial port interrupt from 82C710/711 – also mapped to FIQ device 4
11	IDE hard disc interrupt
12	Floppy disc interrupt from 82C710/711
13	Expansion card interrupt
14	Keyboard serial transmit register empty
15	Keyboard serial receive register full

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

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

Device vectors

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

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

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

Avoiding duplication of drivers

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

Automatic interrupt disabling

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

IRQUtils

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

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

SWI Calls

OS_ClaimDeviceVector (SWI &4B)

Claims a device vector

On entry

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

On exit

R0 - R4 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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

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

If R0 = 8 or 13 then the device driver routine is for an expansion card. R3 gives the address where the expansion card's interrupt status is mapped into memory – see page 6-96 onwards of the chapter entitled *Expansion Cards and Extension* ROMS. Your device driver is called if the IOC chip receives an interrupt from an expansion card, and (LDRB [R3] AND R4) is non-zero.

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

Related SWIs

OS_ReleaseDeviceVector (SWI &4C)

Related vectors

None

OS_ReleaseDeviceVector (SWI &4C)

Releases a device vector

On entry

R0 = device number
R1 = address of device driver routine
R2 = R12 value
R3 = interrupt location if R0 = 8 or 13 on entry
R4 = interrupt mask if R0 = 8 or 13 on entry

On exit

R0 - R4 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call removes a driver from the list of claimants of a device vector. The device driver is identified by the contents of the registers on entry; R0 - R2 (R0 - R4 if R0 = 8 or 13) must be the same as when the device driver was installed on the vector.

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

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

Related SWIs

OS_ClaimDeviceVector (SWI &4B)

Related vectors

None

Technical details

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

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

How a device driver is called

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

1. The device that needs attention alters the status of its interrupt request pin, which is connected to the IOC chip
2. The corresponding bit of one of the IOC's interrupt status registers is set
3. The IOC's interrupt status registers are ANDed with its interrupt mask registers, and the results put in its interrupt request registers
4. If the result was non-zero (ie the device's bit was set in the mask) then an interrupt is sent to the ARM processor
5. If interrupts are enabled, the ARM saves R15 in R14_irq
6. It then forces IRQ mode by setting the M1 bit and clearing the M0 bit of R15, and disables interrupts by setting the I bit
7. The ARM then forces the PC bits of R15 to &18
8. The instruction at &18 is fetched and executed. It is a branch to the code that RISC OS uses to decode IRQs
9. RISC OS examines the interrupt request registers of the IOC chip to see which device number generated the interrupt
10. If the device number was not 8 or 13 (ie the device was not an expansion card) then RISC OS calls the last routine that claimed the corresponding device vector

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

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

11 The device driving routine is executed and returns control.

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

Device driver routines

Entry conditions

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

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

Servicing the interrupt and returning

Your routine must:

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

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

Restrictions

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

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

Speed of execution

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

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

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

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

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

Calling SWIs

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

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

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

3 Control is returned to the interrupt handler.

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

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

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

Recommended procedure

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

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

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

Error handling

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

Re-entrancy

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

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

Documentation of re-entrancy

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

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

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

Clearing interrupt conditions

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

Fast interrupt requests

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

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

FIQ devices

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

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

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

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

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

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

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

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

Similarities between FIQs and IRQs

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

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

Differences between FIQs and IRQs

There are three important differences:

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

The default owner

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

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

Obviously only one current FIQ owner is supported.

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

Using FIQs

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

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

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

   To claim from the background, the reason code in R1 must be &47 (Claim FIQ in background). This service call may fail, but this failure does not imply an error – merely that FIQs could not be claimed. You **must** leave your routine to allow the foreground routine to finish using FIQs and release them. You should schedule a later retry; for example with a disc, you would retry next revolution of the disc. If R1 = 0 on return, you successfully claimed the FIQ vector.
2. Set the IOC fast interrupt mask register to &00, to prevent fast interrupts while you are changing the FIQ code.
3. Poke your FIQ handling routine into addresses &1C upwards. You may use memory up to location &100 (ie the last possible instruction is at &FC).
4. Enable FIQ generation from your device.
5. Set the bit corresponding to your device in the IOC fast interrupt mask register.
6. Start your FIQ operation. You must either poll for its completion, or rely on the completion starting the finalise process in the steps below.
7. End your FIQ operation

8 Set the IOC fast interrupt mask register to zero.

9 Disable FIQ generation from your device.

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

How the FIQ vector is called

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

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

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

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

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

5 The ARM saves R15 in R14_fiq

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

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

8 The FIQ handling routine at &1C is entered

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

Disabling interrupts

There will be times when you want to disable interrupts (ie IRQs). You must only do so with great care; and particularly not for long periods of time since this will have various unwanted effects such as stopping the clock, disabling the keyboard, etc.

SWIs provided

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

More advanced cases

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

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

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

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

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

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

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

...                      ; End of write to IOC

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

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

Service Calls

Service_ReleaseFIQ (Service Call &0B)

Release FIQ

On entry

R1 = &0B (reason code)

On exit

R1 = 0 to claim, else preserved to pass on

Use

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

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

See the section entitled *Using* FIQs on page 1-123 for details of other steps to take when claiming or releasing the FIQ hardware vector, and also the chapter entitled *Hardware vectors* on page 1-103 for additional information about the vector.

Service_ClaimFIQ (Service Call &0C)

Claim FIQ

On entry

R1 = &0C (reason code)

On exit

R1 = 0 to claim, else preserved to pass on

Use

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

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

See the section entitled *Using* FIQs on page 1-123 for details of other steps to take when claiming or releasing the FIQ hardware vector, and also the chapter entitled *Hardware vectors* on page 1-103 for additional information about the vector.

Service_ClaimFIQinBackground (Service Call &47)

Claim FIQ in background

On entry

R1 = &47 (reason code)

On exit

R1 = 0 to claim, else preserved to pass on

Use

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

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

Background claims are released by Service_ReleaseFIQ, as before.

See the section entitled *Using FIQs* on page 1-123 for details of other steps to take when claiming or releasing the FIQ hardware vector, and also the chapter entitled *Hardware vectors* on page 1-103 for additional information about the vector.

SWI Calls

OS_IntOn (SWI &13)

Enables interrupts

On entry

No parameters passed in registers

On exit

Registers preserved

Interrupts

Interrupt status is undefined on entry
Interrupts are enabled on exit
Fast interrupt status is unaltered

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call enables interrupts and returns to the caller with the processor mode unchanged.

Related SWIs

OS_IntOff (SWI &14)

Related vectors

None

OS_IntOff (SWI &14)

Disables interrupts

On entry

No parameters passed in registers

On exit

Registers preserved

Interrupts

Interrupt status is undefined on entry
Interrupts are disabled on exit
Fast interrupt status is unaltered

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call disables interrupts and returns to the caller with the processor mode unchanged.

Related SWIs

OS_IntOn (SWI &13)

Related vectors

None

OS_EnterOS (SWI &16)

Sets the processor to SVC mode

On entry

No parameters passed in registers

On exit

Registers preserved

Interrupts

Interrupt status is unaltered
Fast interrupt status is unaltered

Processor mode

Processor is in SVC mode during the routine, and on exit.

Re-entrancy

SWI is re-entrant

Use

This call returns to the caller in SVC mode. This leaves you using the SVC stack. The interrupt states remain unchanged.

Related SWIs

None

Related vectors

None

Hardware addresses

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

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

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

Finding out more

If you need to know more, you can:

- refer to the earlier chapter entitled ARM *Hardware* on page 1-7
- consult the *Acorn* RISC *Machine family Data Manual*. VLSI Technology Inc. (1990) Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-781618-9.
- consult the datasheets for the various peripheral controllers used, available from their manufacturers
- contact Acorn Customer Service.

IOC registers

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

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

Figure 10.1 Typical memory mapping of IOC registers

Control register

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

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

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

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

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

Other devices

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

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

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

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

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

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

11 Events

Introduction

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

Number	Event type
0	Output buffer has become empty
1	Input buffer has become full
2	Character has been placed in input buffer
3	End of ADC conversion on a BBC I/O expansion card
4	Electron beam has reached last displayed line (VSync)
5	Interval timer has crossed zero
6	Escape condition has been detected
7	RS423 error has been detected
8	Econet user remote procedure has been called
9	User has generated an event
10	Mouse buttons have changed state
11	A key has been pressed or released
12	Sound system has reached the start of a bar
13	PC Emulator has generated an event
14	Econet receive has completed
15	Econet transmit has completed
16	Econet operating system remote procedure has been called
17	MIDI system has generated an event
18	Reserved for use by an external developer
19	Internet has generated an event
20	Reserved for use by an external developer
21	Reserved for use by an external developer
22	Reserved for use by DeviceFS
23	Reserved for use by an external developer

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

Enabling and disabling events

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

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

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

Expansion card modules

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

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

Using events

To use event(s), you must first OS_Claim the event vector EventV. See the chapter entitled *Software vectors* on page 1-59 for further details of vectors. You must then call OS_Byte 14 to enable each of the events you wish to use.

The event routine

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

The restrictions which apply to interrupt handlers also apply to event handlers – namely, event routines are entered with interrupts disabled, with the processor in a non-user mode. They may only re-enable interrupts if they disable them again before passing on or intercepting the call, and they must ensure that the processing of one event is completed before they start processing another. The use of certain operating system calls must be avoided. For further details see the section entitled *Restrictions* on page 1-118.

Finishing with events

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

SWI Calls

OS_Byte 13 (SWI &06)

Disables an event

On entry

R0 = 13
R1 = event number

On exit

R0 preserved
R1 = old enable state
R2 corrupted

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call disables an event by decreasing the count of the number of times that event has been enabled. If the count is already zero, it is not altered. The previous enable state of the event is returned in R1:

R1 = 0	previously disabled
R1 > 0	previously enabled

Note that to disable an event totally, you must use OS_Byte 13 the same number of times as you use OS_Byte 14.

Related SWIs

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

Related vectors

EventV, ByteV

OS_Byte 14 (SWI &06)

Enables an event

On entry

R0 = 14
R1 = event number

On exit

R0 preserved
R1 = old enable state
R2 corrupted

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call enables an event by increasing the count of the number of times that event has been enabled. The previous enable state of the event is returned in R1:

R1 = 0	previously disabled
R1 > 0	previously enabled

When you finish using the vector, you should disable it again by calling OS_Byte 13.

Related SWIs

OS_Byte 13 (SWI &06), OS_GenerateEvent (SWI &22)

Related vectors

EventV, ByteV

OS_GenerateEvent (SWI &22)

Generates an event

On entry

R0 = event number
R1 ... = event parameters

On exit

All registers preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

Note that, as usual, the event vector will only be called if the event number given in R0 has previously been enabled using OS_Byte 14.

Related SWIs

OS_Byte 13 and 14 (SWI &06)

Related vectors

EventV

Details of events

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

Output buffer empty event

R0 = 0
R1 = buffer number

This event is generated when the last character has just been removed from an output buffer (e.g. printer buffer, serial port output buffer) which has output empty events enabled, or an attempt is made to remove another character from the buffer once it has been emptied. See the chapter entitled *Buffers* on page 1-153.

Input buffer full event

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

This event is generated when an input buffer (which has input full events enabled) is full and when the operating system tries to enter a character into the buffer but fails. See the chapter entitled *Buffers* on page 1-153.

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

Character input event

R0 = 2
R1 = buffer number (bits 0 - 30) and byte/block operation flag (bit 31):
bit 31 clear ⇒ byte operation – R2 holds byte
bit 31 set ⇒ block operation – R2 points to block of length R3
R2 = byte to be inserted into keyboard buffer (if byte operation)
else R2 = pointer to data inserted (if block operation)
R3 = number of bytes inserted (if block operation)

This event is generated when a key is pressed, independent of the input stream selected. In the case of block transfers you are given pointers to the original data block. See the chapter entitled *Character Input* on page 2-337 for a description of buffer values for the keyboard buffer.

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

ADC end conversion event

R0 = 3
R1 = channel that just converted

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

Vertical sync event

R0 = 4

This event is generated when the electron beam reaches the bottom of the displayed area and is about to start displaying the border colour. This event corresponds to the time when the OS_Byte 19 call returns to you. In low-resolution modes this will be every fiftieth of a second; in modes requiring a multisync monitor it will be more frequent.

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

Interval timer event

R0 = 5

This event is generated when the interval timer, which is a five-byte value incremented 100 times a second, has reached zero. See OS_*Word* 3 (SWI &07) on page 1-404 for details of the interval timer.

Escape event

R0 = 6

This event is generated when either Esc is pressed or when an escape condition is received from the RS423 input port. See the chapter entitled *Character Input* on page 2-337 for a discussion of escape conditions.

RS423 error event

R0 = 7
R1 = pseudo 6850 status register shifted right 1 place
R2 = character received

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

Bit	Meaning when set
5	Parity error
4	Over-run error
3	Framing error

Econet user remote procedure event

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

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

User event

R0 = 9
R1... = values defined by user

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

Mouse button event

R0 = 10
R1 = mouse X co-ordinate
R2 = mouse Y co-ordinate
R3 = button state
R4 = 4 bytes of monotonic centi-second value

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

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

Key up/down event

R0 = 11
R1 = 0 for key up, 1 for key down
R2 = key number
R3 = keyboard driver ID

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

high	0	1	2	3	4	5	6	7
low 0	Esc	'	Home	P	G	C	Alt (R)	Select
1	F1	1	PageUp	[	H	V	Ctrl (R)	Menu
2	F2	2	NumLock	]	J	B	←	Adjust
3	F3	3	/	\	K	N	↓	
4	F4	4	*	Delete	L	M	→	
5	F5	5	#	Copy	;	,	0	
6	F6	6	Tab	PageDown	"	.	.	
7	F7	7	Q	7	Return	/	Enter	
8	F8	8	W	8	4	Shift (R)		
9	F9	9	E	9	5	↑		
A	F10	0	R	-	6	1		
B	F11	-	T	Ctrl (L)	+	2		
C	F12	=	Y	A	Shift (L)	3		
D	Print	`	U	S		CapsLock		
E	ScrollLock	^	I	D	Z	Alt (L)		
F	Break	Insert	O	F	X	Space		

Figure 11.1 Low-level internal key numbers

Where there is some ambiguity, eg the digit keys, it should be clear from referring to the keyboard layout which code refers to which key. The keys are numbered top to bottom, left to right, starting from Esc at the top left corner. 4D is unused on the UK model, but may be used on some other models for an extra key.

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

Sound start of bar event

R0 = 12
R1 = 2
R2 = 0

This event is generated whenever the sound beat counter is reset to zero, marking the start of a bar. See the chapter entitled *The Sound system* on page 5-335 for more details.

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

PC Emulator event

R0 = 13

This event is claimed by the PC Emulator package.

Econet receive event

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

This event is generated when an Econet reception completes. The status returned in R2 will always be 9 (Status_Received). See the chapter entitled *Econet* on page 6-1 for further details.

Econet transmit event

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

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

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

See the chapter entitled *Econet* on page 6-1 for further details.

Econet OS remote procedure event

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

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

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

See the chapter entitled *Econet* on page 6-1 for further details.

MIDI event

R0 = 17
R1 = event code

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

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

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

Internet event

R0 = 19
R1 = event code
R2 = socket descriptor

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

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

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

Device overrun event

Internet receive frame event

Internet transmission status event

12 Buffers

Introduction

The interrupt system on a RISC OS computer makes extensive use of buffers. These act as temporary holding areas for data after you (or a device) generate it, and before a device (or you) consume it. For example, whenever you type a character on the keyboard, that character is stored in the keyboard input buffer by the keyboard interrupt handler, and it remains there until your program is ready to use it.

The buffer manager

The buffer manager is a global buffer managing system used by DeviceFS to provide buffers for the various devices that can be accessed. It provides a set of calls for setting up a buffer, inserting and removing data from a buffer, and removing a buffer. For more details about the buffer manager see the chapter entitled *The Buffer Manager* on page 5-407.

Filing system buffers

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

Use of buffers

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

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

- inserting a character into a buffer
- removing a character
- counting the space in a buffer

- examining the next character without removing it
- purging a buffer (clearing its contents).

All these operations are implemented as OS_Bytes – see below.

The buffer is also purged implicitly when the escape condition is cleared – see the chapter entitled *Character Input* on page 2-337.

Details of buffers

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

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

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

Buffers 4 to 8

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

Data format

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

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

The bytes are listed in the order in which they would be removed using OS_Byte 145 – see page 1-162.

Usually OS_Mouse reads data from the mouse buffer. If none is available, it returns the current state instead. The mouse buffer is 63 bytes long, so 7 entries may be held at once.

OS_Byte calls provided

The OS_Bytes used to control buffers are described below.

They are, in fact, just an interface to the vectored buffer routines described on page 1-80 onwards of the chapter entitled *Software vectors*. Usually, the OS_Bytes are easier to use. However, there are times when it is preferable, or necessary (for example to read the number of bytes free in an input buffer) to use the vectors. They can be called directly using OS_CallAVector.

It is possible to change the operation of the machine by replacing these calls. In particular, you could write a module which OS_Claims all three buffer vectors, then replaces, say, the printer buffer with a much larger one. You would claim the memory for this from the relocatable module area. The module could have its own configuration byte held in CMOS RAM to specify the size of the buffer, which it would claim on initialisation.

OS_Byte 15 (SWI &06)

Flushes all buffers, or the current input buffer

On entry

R0 = 15
R1 = reason code

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

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

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

Related SWIs

OS_Byte 21 (SWI &06)

Related vectors

ByteV

OS_Byte 21 (SWI &06)

Flushes a specified buffer

On entry

R0 = 21
R1 = buffer number

On exit

R0, R1 preserved
R2 corrupted

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call flushes the specified buffer.

Related SWIs

OS_Byte 15 (SWI &06)

Related vectors

ByteV

OS_Byte 128 (SWI &06)

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

On entry

R0 = 128
R1 = reason code

On exit

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

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

On entry	On exit R1 & R2 contain the:
R1 = 7	mouse x position
R1 = 8	mouse y position
R1 = 246	number of bytes in the mouse buffer
R1 = 252	number of bytes free in the printer buffer
R1 = 253	number of bytes free in the RS423 output buffer
R1 = 254	number of bytes in the RS423 input buffer
R1 = 255	number of bytes in the keyboard buffer

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

Related SWIs

None

Related vectors

ByteV, CnpV

OS_Byte 138 (SWI &06)

Inserts a byte into a buffer

On entry

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

On exit

R0 - R2 preserved
C flag = 0 if character inserted
C flag = 1 if buffer was full

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call inserts the byte specified in R2 into the buffer identified by R1. If C=1 on exit, the byte was not inserted as there was no room.

Inserting bytes into the mouse buffer isn't recommended, but if you must, you should be careful to insert all nine bytes with interrupts disabled, to prevent a real mouse transition from entering data into the middle of your data. You must do so as quickly as possible to prevent latency in the interrupt system.

If the current escape character (usually ASCII 27) is inserted, then appropriate action is taken; see the chapter entitled *Character Input* on page 2-337.

Related SWIs

OS_Byte 145 (SWI &06), OS_Byte 152 (SWI &06), OS_Byte 153 (SWI &06)

Related vectors

ByteV, InsV

OS_Byte 145 (SWI &06)

Gets a byte from a buffer

On entry

R0 = 145
R1 = buffer number

On exit

R0, R1 preserved
R2 = byte extracted
C flag = 0 if byte read
C flag = 1 if buffer was empty

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

Related SWIs

OS_Byte 138 (SWI &06), OS_Byte 152 (SWI &06), OS_Byte 153 (SWI &06)

Related vectors

ByteV, RemV

OS_Byte 152 (SWI &06)

Examines the status of a buffer

On entry

R0 = 152
R1 = buffer number

On exit

R0, R1 preserved
R2 = next byte in buffer, or corrupted if buffer was empty
C flag = 0 if bytes were in buffer
C flag = 1 if buffer was empty

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

Related SWIs

OS_Byte 138 (SWI &06), OS_Byte 145 (SWI &06), OS_Byte 153 (SWI &06)

Related vectors

ByteV, RemV

OS_Byte 153 (SWI &06)

Inserts a byte into one of the two input buffers

On entry

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

On exit

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

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

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

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

If the current escape character (usually ASCII 27) is inserted, then appropriate action is taken; see the chapter entitled *Character Input* on page 2-337.

Related SWIs

OS_Byte 138 (SWI &06), OS_Byte 145 (SWI &06), OS_Byte 152 (SWI &06)

Related vectors

ByteV, InsV

13 Communications within RISC OS

Introduction

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

Service calls

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

This call is fully documented on page 1-243 onwards of the chapter entitled *Modules*.

Window manager SWIs

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

These calls are all fully documented in the chapter entitled *The Window Manager* on page 4-83.

UpCalls

The kernel provides the SWI OS_UpCall, which warns applications of particular situations. It calls the vector UpCallV. To use UpCalls, you must either claim the vector and install a routine on it (see the chapter entitled *Software vectors* on page 1-59), or install an UpCall handler (see the chapter entitled *Program Environment* on page 1-277).

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

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

Writing code to handle UpCalls

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

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

OS_UpCall 1 and 2 (SWI &33)

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

On entry

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

On exit

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

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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

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

It calls the UpCall vector.

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

- prompt you to supply the medium with a string built up using:
  1. the medium type string (passed in R6)
  2. the filing system name (obtained by calling XOS_FSControl 33 acting on the value of R1 – see page 3-110 for details)
  3. the medium name (passed in R2)

  for example:

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

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

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

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

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

Related SWIs

OS_UpCall 4 (SWI &33)

Related vectors

UpCallV

OS_UpCall 3 (SWI &33)

Warns your program that a file is being modified

On entry

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

On exit

All registers preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call warns your program that a file is being modified. The reason code in R9 tells you how:

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

258	Opening for update
259	Closing file
520	Renaming file
521	Setting attributes

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

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

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

The following general points apply:

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

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

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

R9 = 0

Saving memory to file

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

R9 = 1

Writing catalogue information

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

R9 = 2

Writing load address only

R1 = pointer to filename
R5 = pointer to end of buffer
R6 = pointer to special field (or 0)

R9 = 3

Writing execution address only

R1 = pointer to filename
R3 = execution address
R6 = pointer to special field (or 0)

R9 = 4

Writing attributes only

R1 = pointer to object name
R5 = attributes
R6 = pointer to special field (or 0)

R9 = 6

Deleting file

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

R9 = 7

Creating empty file

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

R9 = 8

Creating directory

R1 = pointer to directory name
R2 = load address (to be used as timestamp)
R3 = execution address (to be used as timestamp)
R6 = pointer to special field (or 0)

R9 = 257

Creating and opening for update

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

R9 = 258

Opening for update

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

R9 = 259

Closing file

R1 = external handle

R9 = 520

Renaming file

R1 = pointer to current object name
R2 = pointer to desired object name
R3 = execution address
R6 = pointer to current special field (or 0)
R7 = pointer to desired special field (or 0)

R9 = 521

Setting attributes

R1 = pointer to object name
R2 = pointer to attribute string (control character terminated)

Related SWIs

None

Related vectors

UpCallV

OS_UpCall 4 (SWI &33)

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

On entry

R0 = 4 (Media search end)

On exit

R0 preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is made by RISC OS to inform your program that a missing filing medium has been supplied, or that an operation involving one has been cancelled. It is always preceded by call(s) of OS_UpCall 1 or OS_UpCall 2. It calls the UpCall vector.

To use OS_UpCall 4, you must either claim UpCallV and install a routine on the vector, or install an UpCall handler. This call is typically used to remove error messages displayed when OS_UpCall 1 or 2 was first generated.

Related SWIs

OS_UpCall 1 and 2 (SWI &33)

Related vectors

UpCallV

OS_UpCall 6 (SWI &33)

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

On entry

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

On exit

R0 = 0 if UpCall claimed

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is made by a task that wants to sleep until some termination condition is met, signalled by the contents of the poll word becoming non-zero. It is not available in RISC OS 2.0.

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

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

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

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

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

Related SWIs

OS_UpCall 7 (SWI &33)

Related vectors

UpCallV

OS_UpCall 7 (SWI &33)

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

On entry

R0 = 7 (SleepNoMore)
R1 = pointer to poll word (in a global memory area, eg the RMA)

On exit

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

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is made by PipeFS if an open pipe is closed or deleted. It is not available in RISC OS 2.0.

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

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

Related SWIs

OS_UpCall 6 (SWI &33)

Related vectors

UpCallV

OS_UpCall 8 (SWI &33)

Buffer filling

On entry

R0 = 8
R1 = buffer handle
R2 = 0

On exit

All registers preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This is issued when data is inserted into the specified buffer, and the free space becomes less than the specified threshold.

Related SWIs

None

Related vectors

None

OS_UpCall 9 (SWI &33)

Buffer emptying

On entry

R0 = 9
R1 = buffer handle
R2 = -1

On exit

All registers preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This is issued when data is removed from the specified buffer, and the free space becomes greater than or equal to the current threshold.

Related SWIs

None

Related vectors

None

OS_UpCall 10 (SWI &33)

Stream created

On entry

R0 = 10
R1 = device handle
R2 = 0 if created for transmission (else created for reception)
R3 = external handle used for stream
R4 = internal handle used for stream

On exit

All registers preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This is issued when a stream is created. It serves as a broadcast, and all registers should be preserved.

Related SWIs

None

Related vectors

None

OS_UpCall 11 (SWI &33)

Stream closed

On entry

R0 = 11
R1 = device handle
R2 = 0 if closed for transmission (else closed for reception)
R3 = external handle used for stream
R4 = internal handle used for stream

On exit

All registers preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This is issued when a stream is closed. It serves as a broadcast, and all registers should be preserved.

Related SWIs

None

Related vectors

None

OS_UpCall 256 (SWI &33)

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

On entry

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

On exit

R0 = 0 to stop application, else R0 is preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is made just before a new application is going to be started – for example due to a *Run or module command. It calls the UpCall vector.

To use OS_UpCall 256, you must either claim UpCallV and install a routine on the vector, or install an UpCall handler.

One reason to use this call is so that an application can tidy up after itself before a new one starts, eg removing routines from vectors. For more details, see the chapter entitled *Program Environment* on page 1-277.

Another reason to use this UpCall is to prevent an application from starting. If you don't want the application to start, your routine should set R0 to 0, and intercept the call to the vector. This will cause the error `Unable to start application` to be given. Otherwise, you must pass the call on with all registers preserved.

Related SWIs

None

Related vectors

UpCallV

OS_UpCall 257 (SWI &33)

Informs your program that RISC OS would like to move memory

On entry

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

On exit

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

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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

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

Related SWIs

None

Related vectors

UpCallV

Part 2 – The kernel

14 Modules

Introduction

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

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

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

Several SWIs and * Commands are provided by the operating system for handling modules. For example, loading a module file from the filing system.

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

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

or if:

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

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

This chapter describes what is needed to write a module.

Overview

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

Using modules

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

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

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

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

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

Instantiation

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

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

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

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

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

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

```
*RMKill WaveSynth%Base
```

Writing a module

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

Module header

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

- Where to start executing in the module. This is used by languages and applications.
- Where to call initialisation code. This has to be called before all the others.
- Where to call finalisation code. This is called before removing the module. It allows the module to shutdown any hardware it is using and generally tidy up.
- A title for the module.
- A help string. This is used automatically by RISC OS when help is requested.
- Detailed help on * Commands.
- Entry points for * Commands. RISC OS will decode the * Commands and call the right entry point for a command for you.
- A table to convert to and from SWI names and numbers.
- Entry points for all the SWIs in the module.

- The chunk number for the module. This is the number that is the base for SWI numbers. There can be up to 64 SWIs in a module, all offsets from this chunk number. This is the only entry in the header that isn't a pointer.
- Service call entry (see below).

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

Service calls

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

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

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

- An error has occurred. This is called before the error handler, but is only for module's information, not claiming.
- Reset is about to happen/has just happened.

- Filing system re-initialise. This is called when FileSwitch has been re-initialised and this is broadcast to all filing systems that use it to do the same. This is necessary, because otherwise a filing system could get out of sync with the context in FileSwitch.
- A screen mode change has occurred. This means that all modules can be aware of the screen state and re-read VDU variables, for instance.

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

Technical Details

Module initialisation

When RISC OS is started it automatically initialises all modules in the computer. In RISC OS 2.0 it does so in the order it finds modules, omitting any that are unplugged.

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

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

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

   Hence if an expansion card or extension ROM contains a newer version of a module in the main ROM, the kernel initialises that newer version at the point where the main ROM version would have been initialised. This allows main ROM modules to be replaced without any problems associated with initialisation order.
3. The kernel next scans down the list of modules in expansion cards. For each module in this list, the kernel initialises the newest version of that module, but with the hardware address (in R11) corresponding to that of the expansion card.

   If a module is present both in the main ROM and in an expansion card, the kernel therefore initialises the newest version of that module when scanning the main ROM (as above), and then reinitialises the same module when scanning the expansion cards.
4. The kernel finally scans down the list of modules in extension ROMs. For each module in this list, the kernel checks that it is the newest version of that module, and that it has not already been initialised in lieu of a module in the main ROM or on an expansion card. If a module meets both these criteria the kernel initialises it.

Using modules

OS_Module (SWI &1E) is the main application interface to modules. In its description you will find a complete list of its calls and details of each.

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

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

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

*RMEnsure is a command that will check that a given module and version number is loaded into memory, and will try and load it if it is not.

*UnPlug will disable the ROM version of a given module. This is used if an upgraded version of a module is released and can be loaded from a filing system.

Workspace

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

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

The system works on the assumption that the private word is a pointer to workspace claimed in the RMA. It therefore provides suitable default actions on that basis. For example, the system will attempt to free any workspace claimed using this pointer.

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

Note that workspace allocated through XOS_Module will always lie on an address &XXXXXX4. This enables code written for time-critical software (eg sound voice generators and FIQ handlers) to be aligned within the module body.

Errors in module code

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

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

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

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

Module header format

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

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

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

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

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

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

Service calls

Service calls are made from RISC OS to a module to indicate an occurrence of some kind. Some are claimable, and some are intended as broadcasts of the occurrence only. See the description in OS_ServiceCall (SWI &30) for a complete list of all service calls. It is followed by details of each call. Some of these service calls will also be relevant to other parts of this manual that describe modules. For example, there are service calls that are provided explicitly to serve the International module.

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

Module entry points

Start code

Start executing at the start point of code in a module

Offset in header

&00

On entry

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

On exit

Doesn't return unless error occurs.

Interrupts

Interrupts are enabled on entry
Fast interrupts are enabled

Processor Mode

Processor is in USR mode

Re-entrancy

Entry point is not re-entrant

Use

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

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

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

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

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

Initialisation Code

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

Offset in header

&04

On entry

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

On exit

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

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

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

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

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

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

Typical actions are claiming workspace (via OS_Module) and storing the workspace pointer in the private word. Other actions may include linking onto vectors, declaring the module as a filing system, etc.

The module can refuse to be initialised. If an error is generated during initialisation, the system removes the module and any workspace pointed to by its private word from the RMA. Any error should be dealt with by setting R0 to be an error indicator and returning to the module handler with V set.

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

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

On exit, use the link register passed in R14 to return:

```
MOV PC,R14
```

Return V set or clear depending on whether an error has occurred or not. If an error has occurred, it returns R0 as the error indicator.

Finalisation Code

Called before killing the module

Offset in header

&08

On entry

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

On exit

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

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

This is the reverse of initialisation. This code is called when the system is about to kill all instantiations of the module either completely or temporarily whilst it tidies the RMA.

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

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

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

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

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

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

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

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

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

Service call handler

Called when a service call is issued

Offset in header

&0C

On entry

R1 = service number
R12 = pointer to currently preferred instantiation of the module
R13 = a full, descending stack

On exit

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

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC or IRQ mode

Re-entrancy

Entry point is not re-entrant

Use

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

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

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

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

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

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

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

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

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

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

The service call handler is used when a service call is issued or via an OS_Byte 143 (SWI &06) or OS_ServiceCall (SWI &30). The service calls are described in the section on OS_ServiceCall.

Title string

Offset of a null-terminated module name

Offset in header

&10

Use

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

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

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

Used by OS_Module with reason codes Delete, Enter and Reinit. Also printed by the *Modules command.

Help string

Used when *Help prints information from the module

Offset in header

&14

Use

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

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

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

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

Help and command keyword table

Get help on * Commands or enter them

Offset in header

&18

On entry

R12 = pointer to currently preferred instantiation of the module
R13 = pointer to a full descending stack
R14 = return address

On exit

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

Interrupts

Interrupts are enabled on entry
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

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

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

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

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

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

Figure 14.1 Format of entries in help and command keyword table

The code offset is used for commands. A zero entry means that the string has help text only associated with it. The code is entered with R0 pointing at the command tail and R1 set to the number of parameters (as counted by OSCLI, which means space(s) separate parameters except within double quotation marks). You may not overwrite the command tail.

Information word

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

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

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

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

The flags are as follows:

Bit 31 = 1

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

Bit 30 = 1

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

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

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

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

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

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

Bit 29 = 1

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

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

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

Other comments

Other flags should be zero for upwards compatibility. The invalid syntax message is used by OSCLI as the text of an error message. If the parameters, which are given, fall outside the range specified. If a zero offset is given, a default 'Invalid number of parameters' error is given instead.

The help text is used by *Help. If a keyword in the *Help command tail matches the match string, then the help text is pretty-printed using the RISC OS internal token dictionary. Refer to OS_PrettyPrint (SWI &44) for a full list of the token dictionary.

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

SWI chunk base number

The base of chunk numbers for the module

Offset in header

&1C

Use

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

SWI handler code

Called to handle SWIs belonging to the module

Offset in header

&20

On entry

R11 = SWI number modulo Chunk Size (ie 0 - 63)
R12 = private word pointer
R13 = supervisor stack
R14 contains the flags of the SWI caller

On exit

R10 - R12 may be corrupted

Use

```
MOVS PC,R14
```

to return, having altered R14 flags as appropriate (eg setting V for an error).

Interrupts

Interrupts are unchanged on entry
Fast interrupts are enabled

Interrupts should always be enabled if SWI processing will take a long time (say > 20µs) and the routine can cope with IRQs being enabled. The code to enable IRQs is:

```
MVN     Rn, #I_bit      ; = &08000000
TSTP    Rn, PC          ; preserves other flags
```

To disable IRQs explicitly:

```
MOV     Rn, PC
ORR     Rn, Rn, #I_bit
TEQP    Rn, #0
```

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

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

base to (base + SWI chunk size – 1)

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

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

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

See the section entitled SWI *numbers in detail* on page 1-24 for more details on SWI and chunk numbers.

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

```
.SWIentry
        LDR     R12, [R12]                      ; get workspace pointer
        CMP     R11, #(EndOfJumpTable - JumpTable)/4
        ADDCC   PC, PC, R11, LSL #2             ; dispatch if in range
        B       UnknownSWIError                 ; unknown SWI
.JumpTable
        B       MySWI_0
        B       MySWI_1
        ................
        B       MySWI_n
.EndOfJumpTable
.UnknownSWIError
        ADR     R0, errMesg
        ORRS    PC, R14, #Overflow_Flag
.errMesg
        EQUD    &1E6                            ;Same as system message
        EQUS    "Unknown module operation"
        EQUB    0
```

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

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


```
MOVS PC, R14
```

Otherwise alter the link register, for example by executing

```
ORRS PC, R14, #Carry_Flag
```

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

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

SWI decoding table

Pointer to table of SWI names

Offset in header

&24

Use

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

The table format is:

SWI group prefix
Name of 0th SWI
Name of 1st SWI
.
.
.
Name of nth SWI
0 byte to terminate

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

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

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

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

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

SWI decoding code

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

Offset in header

&28

On entry

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

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

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

On exit

R12 preserved

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

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

Interrupts

Interrupts are enabled on entry
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

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

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

SWI Calls

OS_Byte 143 (SWI &06)

Issue module service call

On entry

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

On exit

R0, R1 preserved
R2 = may contain a return argument

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

Related SWIs

OS_ServiceCall (SWI &30)

Related vectors

ByteV

OS_Module (SWI &1E)

Perform a module operation

On entry

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

On exit

R0 preserved
other register states depends on the reason code

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

R0	Meaning
0	Run
1	Load
2	Enter
3	ReInit
4	Delete
5	Describe RMA
6	Claim
7	Free
8	Tidy
9	Clear

10	Insert module from memory
11	Insert module from memory and move into RMA
12	Extract module information
13	Extend block
14	Create new instantiation
15	Rename instantiation
16	Make preferred instantiation
17	Add expansion card module
18	Lookup module name
19	Enumerate ROM modules
20	Enumerate ROM modules with version

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

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

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

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

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

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

Related SWIs

None

Related vectors

None

OS_Module 0 (SWI &1E)

Run

On entry

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

On exit

Does not return unless error occurs

Use

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

Possible errors are 'File not found', 'No room in RMA', 'Not a module', 'Duplicate module refused to die', and 'Module refuses to initialise'.

Related reason codes

1, 2

OS_Module 1 (SWI &1E)

Load

On entry

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

On exit

R0, R1 preserved

Use

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

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

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

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

Possible errors are 'File not found', 'No room in RMA', 'Not a module', 'Duplicate module refused to die', and 'Module refuses to initialise'.

Related reason codes

0, 2

OS_Module 2 (SWI &1E)

Enter

On entry

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

On exit

Does not return unless error occurs

Use

If the module doesn't have a start address, then this call simply returns. If it does, this call resets the supervisor stack, sets user mode and enters the module, hence making it the current application. Any specified instantiation will become the preferred instantiation. The possible error is 'Module not found'.

For a description of how a module is started up as an application, refer to OS_FSControl 2 (SWI &29).

Related reason codes

0

OS_Module 3 (SWI &1E)

Re-Initialise

On entry

R0 = 3 (reason code)
R1 = pointer to module name plus any parameters for initialisation

On exit

R0, R1 preserved

Use

This is equivalent to reloading the module. It is intended for use in forcing modules that have become confused into a sensible state, without having to reload them explicitly from the filing system. The instruction calls the module through its finalise address and deletes any workspace. It then calls it through its initialisation address to reinitialise it. If the module fails to initialise it is removed from the RMA. Possible errors are 'Module not found' and others dependent on the module.

Related reason codes

8, 9

OS_Module 4 (SWI &1E)

Delete

On entry

R0 = 4 (reason code)
R1 = pointer to module name

On exit

R0, R1 preserved

Use

This reason code (and *RMKill) kill off the currently preferred instantiation of the module or the one specified in the name. For example:

```
*RMKill FileCore%Base
```

This calls the module through its finalise address, frees any workspace pointed at by the private word, delinks the module from the module list and frees the space it was occupying. Possible errors are 'Module not found' and others dependent on the module.

Related reason codes

None

OS_Module 5 (SWI &1E)

Describe RMA

On entry

R0 = 5 (reason code)

On exit

R0 preserved
R2 = size of largest block available in bytes
R3 = total amount free in RMA in bytes

Use

This call returns information on the state of the RMA. It does this by calling OS_Heap with the appropriate descriptor.

Related reason codes

6

OS_Module 6 (SWI &1E)

Claim

On entry

R0 = 6 (reason code)
R3 = required size

On exit

R0 preserved
R2 = pointer to claimed block
R3 preserved

Use

This calls the heap manager to claim workspace in the RMA. If it fails and application workspace is not currently being used then it will attempt to reallocate this memory and retry. It returns with V set if it is still unsuccessful. This call is useful for claiming workspace during the module's initialisation, but may also be used from other module entries.

The possible error is 'No room in RMA'.

Related reason codes

5, 7

OS_Module 7 (SWI &1E)

Free

On entry

R0 = 7 (reason code)
R2 = pointer to block

On exit

R0 preserved
R2 preserved

Use

This calls the heap manager to free a block of workspace claimed from the RMA.

The possible error is 'Not a heap block'.

Related reason codes

6

OS_Module 8 (SWI &1E)

Tidy

On entry

R0 = 8 (reason code)

On exit

R0 preserved

Use

This gives each instantiation of all modules in turn, from the end of the module list and working backwards, a non-fatal finalisation call. Instantiations of a particular module are killed in the order they appear on the current instantiation list.

Should any instantiation refuse to die (temporarily), and another module be called, then the module that has already been called with a non-fatal finalisation is re-initialised. If it cannot be re-initialised, then that module is deleted from the system.

The SWI then exits with the original error. If it succeeds, then it collects the RMA together into one large unfragmented block and reinitialises the modules again. Any private words containing pointers to workspace blocks in the RMA are relocated. This should enlarge application space.

Related reason codes

3, 9

OS_Module 9 (SWI &1E)

Clear

On entry

R0 = 9 (reason code)

On exit

R0 preserved

Use

This deals with each module in turn, removing it from the module list and calling it through its finalise address, if it isn't a ROM module. Errors are generated if modules fail to die.

Related reason codes

3, 8

OS_Module 10 (SWI &1E)

Insert module from memory

On entry

R0 = 10 (reason code)
R1 = pointer to start of module

On exit

R0, R1 preserved

Use

This takes a pointer to a block of memory and links it into the module chain, without moving it. Header fields are checked for validity. All duplicate modules are killed. If it is successful, then the module is called at its initialisation entry.

Possible errors are 'Duplicate module refuses to die' and 'Module refuses to initialise'.

The word immediately before the module start (ie at address R1–4) must contain the length of the module in bytes.

Related reason codes

11

OS_Module 11 (SWI &1E)

Insert module from memory and move into RMA

On entry

R0 = (reason code)
R1 = pointer to start of module
R2 = length of module in bytes

On exit

R0 - R2 preserved

Use

This takes a pointer to a block of memory, and checks its header fields for validity. It then kills any duplicate module, copies the block into the RMA, initialises it and links it into the module chain.

Possible errors are 'Duplicate module refuses to die', 'No room in RMA' and 'Module refuses to initialise'.

Related reason codes

10

OS_Module 12 (SWI &1E)

Extract module information

On entry

R0 = 12 (reason code)
R1 = pointer to module, or 0 for first call
R2 = instantiation number, or 0 for all

On exit

R0 preserved
R1 = updated module number
R2 = updated instantiation number
R3 = module base
R4 = private word (usually workspace pointer)
R5 = pointer to instantiation postfix

Use

This returns pointers to modules and the contents of their private word. It searches the list of modules to see if the module pointer given in R1 is valid. If it is valid, the next descriptor in the module chain is referenced, otherwise the first module descriptor is referenced. Information from the referenced descriptor is then returned. The information returned is exactly that printed by the *Modules command.

Specifying the instantiation number and index in the module list allows all module instantiations to be enumerated. Enumeration can be started with 0 in R1 and R2. This call will:

- count down the module list to find the R1th entry; error if list runs out
- count down the instantiation list to R2th entry; error if list runs out
- set up return information

If the module has more instantiations, R2 += 1 else R1 += 1, R2 = 0

Possible errors are 'No more modules' or 'No more instantiations'.

Related reason codes

13

OS_Module 13 (SWI &1E)

Extend block

On entry

R0 = 13 (reason code)
R2 = pointer to workspace block
R3 = change in size in bytes

On exit

R0 preserved
R2 = pointer to new allocated block
R3 preserved

Use

This allows modules to extend workspace blocks claimed in the RMA. It calls OS_Heap with the appropriate descriptor and attempts to enlarge the RMA if this fails.

The possible error is 'No room in RMA'.

Related reason codes

12

OS_Module 14 (SWI &1E)

Create new instantiation

On entry

R0 = 14 (reason code)
R1 = pointer to full name of new instantiation and any parameters for initialisation

On exit

R0, R1 preserved

Use

This creates new instantiations of existing modules, using the syntax:

module_title%instantiation

For example:

```
FileCore%RAM
```

Related reason codes

15, 16

OS_Module 15 (SWI &1E)

Rename instantiation

On entry

R0 = 15 (reason code)
R1 = pointer to current module%instantiation name
R2 = pointer to new postfix string

On exit

R0 - R2 preserved

Use

This renames an existing instantiation of a module. For example:

```
FileCore%RAM
```

to

```
FileCore%ADFS
```

Related reason codes

14, 16

OS_Module 16 (SWI &1E)

Make preferred instantiation

On entry

R0 = 16 (reason code)
R1 = pointer to full module%instantiation name

On exit

R0, R1 preserved

Use

This enables you to select the preferred instantiation of a particular module.

Related reason codes

14, 15

OS_Module 17 (SWI &1E)

Add expansion card module

On entry

R0 = 17 (reason code)
R1 = pointer to environment string
R2 = chunk number
R3 = ROM section

On exit

R0 - R3 preserved

Use

This allows expansion card and extension ROM modules to be added to the module list. Note that extension ROMs are not supported in RISC OS 2.0.

Valid ROM sections are:

ROM section	Meaning	
-1	System ROM	
0	Expansion card 0	
1	Expansion card 1	
2	Expansion card 2	
3	Expansion card 3	
-2	System ROM 1	(not in RISC OS 2.0)
-3	System ROM 2	(not in RISC OS 2.0)
-4	System ROM 3 (etc)	(not in RISC OS 2.0)

Related reason codes

10

OS_Module 18 (SWI &1E)

Look-up module name

On entry

R0 = 18 (reason code)
R1 = pointer to full *module_title%instantiation* name

On exit

R0 preserved
R1 = module number
R2 = instantiation number
R3 = pointer to module code
R4 = private word contents
R5 = pointer to postfix string

Use

This returns pointers to modules and the contents of their private word. It searches the list of modules to see if the module pointer given in R1 is valid. If it is valid, the module descriptor is referenced. Information from the referenced descriptor is then returned.

Related reason codes

12, 19, 20

OS_Module 19 (SWI &1E)

Enumerate ROM modules

On entry

R0 = 19 (reason code)
R1 = module number (0 to start full enumeration)
R2 = ROM section (–1 to start full enumeration)

On exit

R0 preserved
R1 = module number of found module + 1
R2 = ROM section of found module
R3 = pointer to module name
R4 = –1 unplugged
0 inserted but not in the module chain ie dormant
1 active
2 running
R5 = chunk number of expansion card module

Use

This call returns information on one module that is currently in ROM, along with its status. The module found is the given number of modules on from the start of the given ROM section. If there are insufficient modules in the ROM section then the search continues with the next section; so the fifth module in a four module section would in fact be the first module of the next section.

The ROM sections are scanned in this order:

ROM section	Meaning	
–1	System ROM	
0	Expansion card 0	
1	Expansion card 1	
2	Expansion card 2	
3	Expansion card 3	
–2	System ROM 1	(not in RISC OS 2.0)
–3	System ROM 2	(not in RISC OS 2.0)
–4	System ROM 3 (etc)	(not in RISC OS 2.0)

The values returned in R0 - R2 are the correct ones to use this call to enumerate the next module, hence repeated calls will give a full enumeration of all ROM modules.

The call returns the error 'No more modules' (error number &107) if there are no more modules from the point specified in the ordering.

Related reason codes

12, 18, 20

OS_Module 20 (SWI &1E)

Enumerate ROM modules

On entry

R0 =20 (reason code)
R1 = module number (0 to start full enumeration)
R2 = ROM section (-1 to start full enumeration)

On exit

R0 preserved
R1 = module number of found module + 1
R2 = ROM section of found module
R3 = pointer to module name
R4 = -1 unplugged
0 inserted but not in the module chain ie dormant
1 active
2 running
R5 = chunk number of expansion card module
R6 = BCD version number (derived from module's help string)

Use

This call returns information on one module that is currently in ROM, along with its status. The call is identical to OS_Module 19, except that on exit R6 holds a BCD (binary coded decimal) form of the module's version number, as derived from the module's help string. The top 16 bits of this value hold the integer part of the version number, and the bottom 16 bits hold the fractional part: eg if the version number of the module is '3.14' then the value returned would be &00031400.

The module found is the given number of modules on from the start of the given ROM section. If there are insufficient modules in the ROM section then the search continues with the next section; so the fifth module in a four module section would in fact be the first module of the next section.

The ROM sections are scanned in this order:

ROM section	Meaning	
-1	System ROM	
0	Expansion card 0	
1	Expansion card 1	
2	Expansion card 2	
3	Expansion card 3	
-2	System ROM 1	(not in RISC OS 2.0)
-3	System ROM 2	(not in RISC OS 2.0)
-4	System ROM 3 (etc)	(not in RISC OS 2.0)

The values returned in R0 - R2 are the correct ones to use this call to enumerate the next module; hence repeated calls will give a full enumeration of all ROM modules.

The call returns the error 'No more modules' (error number &107) if there are no more modules from the point specified in the ordering.

Related reason codes

12, 18, 19

Service Calls

OS_ServiceCall (SWI &30)

Issue a service call to a module

On entry

R1 = service number
other registers are parameters and depend upon the service number

On exit

R1 = 0 if service was claimed, preserved otherwise
other registers up to R8 may be modified if the service was claimed

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ServiceCall is used to issue a service call. It can be used by any program (including a module) which wishes to pass a service around the current module list. For example, someone wishing to use FIOs might issue the claim/release service calls.

Here is a list of the available service calls, the details of which can be found on the appropriate pages:

No	Name	Meaning	on page
&00	Service_Serviced	Service call claimed	page 1-247
&04	Service_UKCommand	Unknown command	page 1-248
&06	Service_Error	Error has occurred	page 1-249
&07	Service_UKByte	Unknown OS_Byte	page 1-250
&08	Service_UKWord	Unknown OS_Word	page 1-251
&09	Service_Help	*Help has been called	page 1-252
&0B	Service_ReleaseFIQ	Release FIQ	page 1-126
&0C	Service_ClaimFIQ	Claim FIQ	page 1-127
&11	Service_Memory	Memory controller about to be remapped	page 1-350
&12	Service_StartUpFS	Start-UpFiling System	
&18	Service_Post_Help	End of *Help code	page 1-253
&27	Service_Reset	Post-Reset	page 4-137
&28	Service_UKConfig	Unknown *Configure	page 1-254
&29	Service_UKStatus	Unknown *Status	page 1-255
&2A	Service_NewApplication	Application about to start	page 1-256
&40	Service_FSRedeclare	Filing system re-initialise	
&41	Service_Print	For internal use only	page 5-175
&42	Service_LookupFileType	Lookup file type	page 1-257
&43	Service_International	International service	page 5-257
&44	Service_Keyhandler	Keyboard handler	page 2-356
&45	Service_PreReset	Pre-reset	page 6-104
&46	Service_ModeChange	Mode change	page 2-128
&47	Service_ClaimFIQinBackground	Claim FIQ in background	page 1-128
&48	Service_ReAllocatePorts	Econet restarting	page 6-30
&49	Service_StartWimp	Start the Wimp	page 4-138
&4A	Service_StartedWimp	Started the Wimp	page 4-140
&4B	Service_StartFiler	Start the Filer	page 3-466
&4C	Service_StartedFiler	Started the Filer	page 3-468
&4D	Service_PreModeChange	Mode change	page 2-129
&4E	Service_MemoryMoved	OS_ChangeDynamicArea has just finished	page 1-351
&4F	Service_FilerDying	Filer is dying	page 3-469
&50	Service_ModeExtension	Allow soft modes	page 2-131
&51	Service_ModeTranslation	Translate modes for unknown monitor types	page 2-133
&52	Service_MouseTrap	For non-click mouse warnings	page 4-146
&53	Service_WimpCloseDown	Trap WimpCloseDown calls	page 4-147
&54	Service_Sound	Parts of the Sound system starting/dying	page 5-349
&55	Service_NetFS	Either a *Logon or a *Bye has occurred	page 3-327

&56	Service_EconetDying	Econet about to leave	page 6-31
&57	Service_WimpReportError	Wimp is opening/closing a ReportError window	page 4-149
&59	Service_ResourceFSStarted	Comes after ResourceFSStarting (for clients)	page 3-392
&5A	Service_ResourceFSDying	ResourceFS is killed/reloaded	page 3-393
&5B	Service_CalibrationChanged	Screen calibration is changed	page 4-390
&5C	Service_WimpSaveDesktop	Save desktop to a file request	page 4-150
&5D	Service_WimpPalette	Palette change	page 4-151
&5E	Service_MessageFileClosed	From message file module	page 5-237
&5F	Service_NetFSDying	NetFS is dying	page 3-328
&60	Service_ResourceFSStarting	Starting ResourceFS	page 3-394
&64	Service_TerritoryManagerLoaded	Territory manager started	page 5-282
&65	Service_PDriverStarting	PDriver sharer module starting up	page 5-176
&66	Service_PDumperStarting	PDumper module starting up	page 5-177
&67	Service_PDumperDying	PDumper module dying	page 5-178
&68	Service_CloseFile	Close a file	
&69	Service_IdentifyDisc	Identify disc format	
&6A	Service_EnumerateFormats	Format sub-menu entries	
&6B	Service_IdentifyFormat	Identify format	
&6C	Service_DisplayFormatHelp	Display list of available formats	
&6D	Service_ValidateAddress	Invalid address range called	page 1-352
&6E	Service_FontsChanged	New FontSPath detected	page 5-13
&6F	Service_BufferStarting	Allows modules to register buffers with the buffer manager	page 5-409
&70	Service_DeviceFSStarting	DeviceFS is starting	page 3-404
&71	Service_DeviceFSDying	DeviceFS is dying	page 3-405
&72	Service_SwitchingOutputTo Sprite	Output switched to sprite, mask or screen	page 2-261
&73	Service_PostInit	all modules have been initialised	page 1-258
&75	Service_TerritoryStarted	New territory starting	page 5-283
&76	Service_MonitorLead Translation	Translate monitor lead ID	page 2-134
&77			
&78	Service_PDriverGetMessage	Get common messages file	page 5-179
&79	Service_DeviceDead	Device has been killed by DeviceFS	page 3-406
&7A	Service_ScreenBlanked	Screen blanked by screen blanker	page 5-430
&7B	Service_ScreenRestored	Screen restored by screen blanker	page 5-431
&7C	Service_DesktopWelcome	Desktop starting	page 4-152
&7D	Service_DiscDismounted	Disc dismounted	page 3-472
&7E	Service_ShutDown	Switcher shutting down	page 4-153
&7F	Service_PDriverChanged	PDriver has changed	page 5-180

&80	Service_ShutdownComplete	Shutdown completed	page 4-154
&81	Service_DeviceFSCloseRequest		page 3-407
&82			
&FE	Service_Tube	Internal use only	page 1-259
&10800	Service_ADFSPodule		page 6-105
&10801	Service_ADFSPoduleIDE		page 6-106
&10802	Service_ADFSPoduleIDEDying		page 6-107

Related SWIs

OS_Byte 143 (SWI &06)

Related vectors

None

Service_Serviced (Service Call &00)

Service call has been claimed

On entry

R1 = 0

On exit

R1 preserved

Use

This call is passed around following a successful claiming of a service call by a module.

Service_UKCommand (Service Call &04)

Unknown command

On entry

R0 = pointer to command
R1 = &04 (reason code)

On exit

R0 = 0 for no error, else error pointer
R1 = 0 to claim the command, or preserved to pass on

Use

If you claim the call and execute the command successfully you should set R1 to 0. If an error occurs during execution then you should return with the pointer to the error buffer in R0. This call is issued after OSCLI has searched modules but before the filing system is called to try to *Run. It is also used to implement NetFS file server commands.

Note that this is the 'historical' way of dealing with unknown commands. You should, in preference, use the command string entry point.

Service_Error (Service Call &06)

Error has occurred

On entry

R0 = pointer to error block
R1 = &06 (reason code)

On exit

R0 preserved
R1 preserved to pass on (must never be claimed)

Use

This call is issued after an error has occurred but before the error handler is called. It is included 'for your information', and must not be claimed.

Service_UKByte (Service Call &07)

Unknown OS_Byte

On entry

R1 = &07 (reason code)
R2 = OS_Byte number
R3 = first parameter
R4 = second parameter

On exit

R1 = 0 to claim, else preserved to pass on
R3 = value to return in R1 to caller
R4 = value to return in R2 to caller
Errors cannot be returned

Use

If the OS_Byte number is one used by the module it is passing through, you should execute it and claim the call by setting R1 to zero.

If you don't recognise the OS_Byte number, pass the call on by returning with the registers preserved.

Service_UKWord (Service Call &08)

Unknown OS_Word

On entry

R1 = &08 (reason code)
R2 = OS_Word number
R3 = pointer to OS_Word parameter block

On exit

R1 = 0 to claim, else preserved to pass on
Errors cannot be returned

Use

If the OS_Word number is one used by the module it is passing through, you should execute it and claim the call by setting R1 to zero.

If you don't recognise the OS_Word number, pass the call on by returning with the registers preserved.

Service_Help (Service Call &09)

*Help has been called

On entry

R0 = pointer to command
R1 = &09 (reason code)

On exit

R0 preserved
R1 = 0 to claim, else preserved to pass on

Use

This is issued at the start of *Help. You should claim this call only if you wish to replace *Help completely. The usual way for a module to provide help is through its help text table.

Service_Post_Help (Service Call &18)

Passed round at the end of *Help code

On entry

On exit

Use

Service_UKConfig (Service Call &28)

Unknown *Configure

On entry

R0 = pointer to command tail, or 0 if none given
R1 = &28 (reason code)

On exit

R0 = < 0 for no error,
small integer for errors described below,
or error pointer for other errors
R1 = 0 if configure option recognised and no error, else preserved to pass on

Use

If R0 = 0 on entry, you should print your *Configure syntax line(s), if any, and exit with registers preserved.

If R0 ≠ 0, then R0 is a pointer to the command tail. If you decode the command tail, and recognise it, you should claim the call by setting R1 to 0. If an error is detected, should also return with V set and return the error in R0 as follows:

Value	Meaning
0	Bad *Configure option
1	Numeric parameter needed
2	Parameter too large
3	Too many parameters
>3	R0 is an error pointer returned by *Configure

If you don't recognise the command tail, you should exit with registers preserved.

Note that it is also possible to trap unknown *Configure commands through the module's command table (see the section entitled *Help and command keyword table* on page 1-207) – which is the preferred method. Only one of these mechanisms should be used.

Service_UKStatus (Service Call &29)

Unknown *Status

On entry

R0 = pointer to command tail, or 0 if none given
R1 = &29 (reason code)

On exit

R0 preserved
R1 = 0 is status option recognised and no error, else preserved to pass on

Use

If R0 = 0, you should list your status(es) and pass on the service call.

If R0 ≠ 0, then R0 is a pointer to the command tail. If you decode the command tail, and recognise it, you should print the associated information and claim the call. Otherwise you should not claim the call.

Note that it is also possible to trap unknown *Status commands through the module's command table (see the section entitled *Help and command keyword table* on page 1-207) – which is the preferred method. Only one of these mechanisms should be used.

Service_NewApplication (Service Call &2A)

Application about to start

On entry

R1 = &2A (reason code)

On exit

R1 = 0 to prevent application from starting, else preserved to pass on

Use

This service is called when an application is about to start due to a *Go, *RMRun or *Run-type operation. If you don't want the application to start, you should claim the call, otherwise pass it on.

Service_LookupFileType (Service Call &42)

Lookup file type

On entry

R1 = &42 (reason code)
R2 = file type (in lower three nibbles)

On exit

R1 = 0 if the module knows the file type, else preserved to pass on
R2 = first four characters, if known, else preserved
R3 = last four characters, if known, else preserved

Use

This call is passed round when FileSwitch would like to convert a twelve-bit file type into a textual name. If the file type passed in R2 is known to you, you should return with R1=0, and R2, R3 containing the eight characters in the name. If no-one claims the call, FileSwitch will convert the number into a three-digit hex value padded with spaces. This might be loaded as follows:

Service_PostInit (Service Call &73)

All modules have been initialised

On entry

On exit

This service call should not be claimed.

Use

This is issued on a reset, after all the ROM resident modules (including those on 5th column ROM and Podules) have been initialised.

Service_Tube (Service Call &FE)

Internal use only

*Commands

*Modules

Displays information about all installed relocatable modules

Syntax

```
*Modules
```

Parameters

None

Use

*Modules displays information about all relocatable modules which are currently installed in the machine.

The command displays the number allocated to each module, its position in memory, the address of its workspace, and its name.

- The number may change as other modules are installed and removed.
- The names listed by this command are the module titles, which are used as parameters for other commands such as *RMKill.

Example

```
*Modules
No. Position Workspace Name
  1 0380BED8 00000000  UtilityModule
  2 038251A8 01800014  Podule
    -
    -
 81 039EAF10 00000000  !Edit
 82 039F17E4 0181E984  DOSFS
```

Related commands

*ROMModules

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMClear

Deletes all relocatable modules from the module area

Syntax

```
*RMClear
```

Parameters

None

Use

*RMClear deactivates all relocatable modules in the module area, deletes them, and frees their workspace. Use this command only with extreme caution, as it is so drastic in its effects.

ROM resident modules are not affected by *RMClear; if you wish to disable such a module, you should use *RMKill or *Unplug.

Related commands

*RMKill, *RMReinit, *RMTidy, *Unplug

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMEnsure

Checks the presence and version of a module

Syntax

```
*RMEnsure module_title version_number [command]
```

Parameters

module_title	the title of any currently installed module
version_number	a number against which the version number will be checked
command	a Command Line command

Use

*RMEnsure checks that a module is present and is the given version (or a more recent one). A command, optionally given as a third parameter, is executed if this is not the case. *RMEnsure is usually used in command scripts or programs to ensure that modules they need are loaded and of a recent enough version.

Example

```
*RMEnsure WindowManager 2.01 *RMLoad System:Wimp
```

Related commands

None

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMFaster

Makes a module faster by copying it from ROM to RAM

Syntax

`*RMFaster module_title`

Parameters

module_title	the title of any ROM resident module

Use

*RMFaster makes a copy of a ROM resident relocatable module and places it in RAM. The module will run faster because RAM can be accessed faster than ROM.

Example

```
*RMFaster BASIC
```

Related commands

None

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMInsert

Reverses the action of a previous *Unplug command

Syntax

`*RMInsert module_title [ROM_section]`

Parameters

module_title	the title of any ROM resident module
ROM_section	ROM section to restrict command to

Use

*RMInsert reverses the action of a previous *Unplug command, but without reinitialising any modules.

If no ROM section number is specified, then this command clears the unplug bit for all versions of the specified module present in the machine.

If a ROM section number is specified, then this command clears the unplug bit for all versions of the specified module present in the given section. ROM section numbers are:

ROM section	Meaning
-1	System ROM
0	Expansion card 0
1	Expansion card 1
2	Expansion card 2
3	Expansion card 3 (etc)
-2	System ROM 1
-3	System ROM 2
-4	System ROM 3 (etc)

This command is not available in RISC OS 2.0.

Example

```
*RMInsert MIDI 1
```

Related commands

*RMReInit, *Unplug

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMKill

Deactivates and deletes a relocatable module

Syntax

*RMKill *module_title*[%*instantiation*]

Parameters

module_title	the title of any currently installed module
instantiation	the instantiation of any currently installed module

Use

*RMKill deactivates the preferred instantiation of a relocatable module (or the specified instantiation if the second argument is used) and releases its workspace. If the module is in RAM, it is also deleted. If it is ROM resident, it is made inactive until reinitialised by the *RMReInit command, or until the next hard reset. Use this command only with extreme caution, as it may be drastic in its effects.

Example

```
*RMKill Debugger
```

Related commands

*RMClear, *RMReInit, *RMTidy, *Unplug

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMLoad

Loads and initialises a relocatable module

Syntax

*RMLoad *filename* [*module_init_string*]

Parameters

filename	a valid pathname specifying a module file
module_init_string	optional parameters to the module

Use

*RMLoad loads and initialises a relocatable module. It can then be accessed by the help system, and can provide SWIs and * Commands if available.

The file must have file type FFA, otherwise the module handler will refuse to load it.

The optional initialisation string can be used to pass parameters to certain modules so they initialise themselves in a particular way. For example, you might use it to specify the amount of workspace that the module should claim, or a file that the module should load.

Example

```
*RMLoad WaveSynth $.Waves.Brass14
```

Related commands

*RMRun

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMReInit

Reinitialises a relocatable module

Syntax

*RMReInit *module_title* [*module_init_string*]

Parameters

module_title	the title of any currently installed module, active or otherwise
module_init_string	optional parameters to the module

Use

*RMReInit reinitialises a relocatable module, reversing the action of any previous *RMKill or *Unplug command. The module is returned to the state it was in when it was loaded. Use this command only with extreme caution, as it may be drastic in its effects.

- If the specified module is active, then it is killed and then re-initialised.
- If the specified module is not active, but is in the ROM, then the unplug bit in CMOS RAM is cleared for all versions of the specified module, and then the newest version of the module is initialised. (Under RISC OS 2.0 it is the first found version that is initialised.)

The optional initialisation string can be used to pass parameters to certain modules so they reinitialise themselves in a particular way. For example, you might use it to specify the amount of workspace that the module should claim, or a file that the module should load.

This command can produce unexpected results, for a variety of reasons. For example:

- The order of module initialisation is important in RISC OS. If a module relies on a second module being later initialised, you cannot successfully reinitialise the first module without then reinitialising the second.
- Under the desktop, a reinitialised module does not get restarted as a task unless you re-enter the desktop.

Example

```
*RMReInit Debugger
```

Related commands

*RMClear, *RMKill, *RMTidy, *Unplug

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMRun

Runs a relocatable module

Syntax

`*RMRun filename`

Parameters

filename	a valid pathname specifying a module file

Use

*RMRun loads and initialises a relocatable module. It can then be accessed by the help system, and can provide SWIs and * Commands if available. The module is then run, if it can be.

This call is equivalent to a call to *RMLoad followed by an enter operation in OS_Module. If the module is already resident, then it will simply be entered.

If a module cannot be run, then this command is equivalent to a *RMLoad command.

The file must have file type FFA, otherwise the module handler will refuse to load it.

Example

```
*RMRun My_Module
```

Related commands

*RMLoad

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMTidy

Compacts the module area and reinitialises all the modules it contains

Syntax

```
*RMTidy
```

Parameters

None

Use

*RMTidy collects together free space in the module area by moving and reinitialising all the modules it contains. The free space is gathered into a consecutive chunk of memory.

Use this command only with extreme caution, as it is so drastic in its effects.

Related commands

*RMClear

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*ROMModules

Displays information about all relocatable modules currently installed in ROM

Syntax

```
*ROMModules
```

Parameters

None

Use

*ROMModules displays information about all relocatable modules which are currently installed in ROM.

The command displays the number allocated to each module, whether it is part of the system or in expansion cards or in an extension ROM, its name, and its status: active, running, dormant or unplugged. (Note that RISC OS 2.0 does not support extension ROMs, nor does it give a version number or report modules as running.)

- The names listed by this command are the module titles, which are used as parameters for other commands such as *RMKill.

System modules are stored in ROM, but may still be *RMKilled, *Unplugged, or replaced by RAM-based modules.

Example

```
*ROMModules
No. Position    Module name             Version Status
  1 System ROM  UtilityModule           2.20    Active
  2 System ROM  Podule                  1.23    Active
  3 System ROM  FileSwitch              1.98    Active
  4 System ROM  ResourceFS              0.09    Active
  5 System ROM  Messages                0.16    Active
        ...
  1 Podule 1    Support16a              1.00    Active
        -
  1 Extn ROM 1  Tube6502Emulator        1.17    Dormant
  1 Extn ROM 2  Turbo6502Emulator       1.17    Dormant
  1 Extn ROM 3  Tube6502Emulator        1.17    Active
  2 Extn ROM 3  Turbo6502Emulator       1.17    Active
  1 Extn ROM 4  FontManager             2.85    Active
        ...
```

or under RISC OS 2:

```
*ROMModules
No. Position    Module Name             Status
  1 System ROM  UtilityModule           Active
  2 System ROM  FileSwitch              Active
  3 System ROM  Desktop                 Active
        -
  1 Podule 0    MailBleep               Dormant
  2 Podule 0    ROMBoard                Dormant
        ...
```

Related commands

*Modules

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*Unplug

Kills and disables all copies of a ROM resident module

Syntax

*Unplug [*module_title* [*ROM_section*]]

*Unplug [*module_title*] (RISC OS 2.0)

Parameters

module_title	the title of any ROM resident module
ROM_section	ROM section to restrict command to – this parameter is not recognised by RISC OS 2.0

Use

*Unplug kills all copies of the named ROM module, releasing any workspace used. (In RISC OS 2.0 only the first copy found is deleted.) It also disables all versions of that module – whether in the system ROM, expansion cards or extension ROMs – by preventing them from being initialised (and hence available for use). This setting is stored in the CMOS RAM, and so is permanent even across a reset. To enable the module again you must use the *RMReinit or *RMInsert command. (The latter command is not available in RISC OS 2.0.)

If you supply a ROM section parameter, *Unplug restricts its effects to modules that are in that ROM section. ROM section numbers are:

ROM section	Meaning
-1	System ROM
0	Expansion card 0
1	Expansion card 1
2	Expansion card 2
3	Expansion card 3
-2	System ROM 1
-3	System ROM 2
-4	System ROM 3 (etc)

You should use this command with caution, otherwise you may find programs stop working because you have unplugged a module that is essential to them.

If no parameters are given, the unplugged ROM modules are listed.

Example

```
*Unplug RAMFSFiler        disables the RAMFSFiler module
```

Related commands

*RMInsert, *RMKill, *RMReInit

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

15 Program Environment

Introduction

The program environment refers to the conditions under which a program or module executes. There are three aspects to this environment.

- The memory used by the code and allocated for transient workspace.
- The handlers used by a program or module. A handler is a piece of code called when certain conditions occur. RISC OS provides a set of default handlers, so that something valid will occur. Here is a brief list of the kinds of conditions that we are talking about:
  - an error
  - an escape condition
  - an event
  - certain hardware exceptions, such as an undefined instruction
  - a break point
  - an unused SWI is called
  - when a program or module terminates.
- The system variables are a textual way of finding information about various aspects of the system. There are several kinds of variables:
  - string variables which contain characters only
  - integer variables which contain an integer
  - macro variables which are like string variables, except that they can contain references to special characters and other system variables.

Overview and Technical Details

Starting a task

There are several ways of executing a piece of code. You can:

- *RMRun the program
- OS_Module 'Enter' a module
- *Run the program
- *Go, to execute the program in memory

Modules

The first two are described in the chapter entitled *Modules*. They are really the same thing. When a file is *RMRun, it is loaded into the relocatable module area. Its initialisation code is called, so that it can claim workspace etc, then its start code is called.

A module can also cause its own start entry point to be called if it wants to become the current application, using OS_Module. BASIC is an example of this. The *BASIC command is recognised by the OS using the BASIC module's * Command table. The OS calls the routine which handles the *BASIC command, and this routine calls OS_Module with the reason code 'enter'. For details on calling modules see the chapter entitled *Modules* on page 1-191.

Programs on file

The third case applies to files which have no file type, or have type FF8. In the first case, the file is loaded at its load address, then it is started as an application through its execution address. If the file type is FF8, the file is loaded at &8000 and started as an application there.

Programs in memory

Finally, if you call a machine code program using the *Go command, it becomes the current application. (This implies that you shouldn't use *Go to call RAM-based routines from a language, as the routine can't return – R14 contains no return address at this point.)

In all of these cases, the program is called in user mode, with interrupts enabled. Where a module is called, R12 points to the module's private word.

Transient programs

A file with type FFC (utility) must contain position independent code. When such a file is *RUN, it is loaded into the RMA and executed. This is used when you want to run a utility and then return to the program environment that you were in before running it. On entry to a transient program, registers are as follows:

R0 = pointer to command line
R1 = pointer to command tail
R12 = pointer to workspace
R13 = pointer to workspace end (stack)
R14 = return address
User mode, interrupts enabled

The workspace is 1024 bytes long, in the location given by R12 and R13 on entry. If more is required, it may be allocated from the RMA. The utility should return using MOV PC,R14 (freeing any extra workspace first). It does not become the current application and must not call OS_Exit.

Note that R0 points to the first character of the command name, and R1 points to the first character of the command tail (with spaces skipped). This will be a control character if there were no parameters.

When a utility returns, the space it occupies is freed. Utilities are nestable – you can execute one utility from within another.

Note that utilities are viewed as system extensions. This means that they must only use the X form SWIs, so that the error handler is not called by their actions. A utility can return with an error by setting V and pointing R0 at an error block as usual.

Ending a task

Before describing the calls which control the application program's environment, it is worth explaining how to leave an application. In general, a simple 'return from subroutine' using MOV PC,R14 won't suffice. Instead, you should use a routine called OS_Exit (SWI &11). This passes control back to a well-defined place, which defaults to the supervisor * prompt, but could equally be a location in the previous application.

*Quit is equivalent to a call to OS_Exit.

OS_ExitAndDie (SWI &50) is like OS_Exit, but will kill a named module as well. This is used when a module is specific to a particular application.

System variables

The system variables, maintained by the operating system in the system heap, provide a convenient way by which programs can communicate. Variables are accessed by their textual name. The name may contain any non-space, non-control character. When a variable is created, the case of the letters is preserved. However, when names are looked up the case is ignored, and you can use the characters '#' and '*' – just like looking up filenames.

Naming

You should avoid the use of wholly numeric names for system variables, such as 123, as this causes difficulties when the GS string operations are used to look up a variable's contents. In particular, they will always take <123> to mean the ASCII code 123, and will not attempt to look up the name as a variable. See the chapter entitled *Conversions* on page 1-429 for details of the GS calls, specifically OS_GSRead and OS_GSTrans.

Types

There are several types of system variable:

- String variables can contain any characters you like; these are returned when the string is read. They can be set with *Set.
- Integer variables are four-byte signed integers. They can be set with *SetEval or *SetMacro.
- Macros are strings that are passed through OS_GSTrans when the string is read. This means that if the macro contains references to variables or other OS_GSReadable items, the appropriate translation takes place whenever the variable is accessed. They can be set with *SetMacro.

  A classic example of using a macro is to set the command line prompt CLI$Prompt to the current time using:

  ```
  *SetMacro CLI$prompt <Sys$Time><&20>
  ```

  Every time the prompt is displayed, it shows the current time, followed by a space.
- The final type of variable is machine code routines. A routine is called whenever the variable is to be read, and another when it is set. This allows great flexibility in the way in which such variables behave. For example, you could make a variable directly control a CMOS RAM location using this technique. Sys$Time is a good example of a code variable.

All the above types can be set with OS_SetVarVal (SWI &24) and read with OS_ReadVarVal (SWI &23).

Any non-code variable can be removed using *Unset. *Show will list the setting of one or more variables.

Miscellaneous environment features

OS_GetEnv (SWI &10) is a multi-purpose SWI that provides three useful pieces of information:

1. The address of the * Command string.
   This can be processed with OS_ReadArgs, which is described on page 1-453 of the chapter entitled *Conversions*.
2. The real time that the program was started.
3. The maximum amount of memory allocated for the program.
   This can be altered with reason code 0 of OS_ChangeEnvironment.

OS_WriteEnv (SWI &48) allows you to set the program start time and the command string.

Handlers

Handlers are short routines used to cope with special conditions that can occur under RISC OS. Here is a complete list of the handlers:

Handler
Undefined instruction
Prefetch abort
Data abort
Address exception
Error
CallBack
BreakPoint
Escape
Event
Exit
Unused SWI
UpCall

All of the calls that install user handlers pass through ChangeEnvironmentV. This can be intercepted to stop a subprogram changing parts of the environment that its parent wants to keep: for example, a debugger.

Before reading this section, you should be familiar with the chapters entitled *Software vectors* on page 1-59 and *Hardware vectors* on page 1-103, since many of these handlers are directly called from these vectors.

SWIs

OS_ChangeEnvironment (SWI &40) is the central SWI for handlers. There are several other routines that perform subsets of its actions. You are strongly recommended to use OS_ChangeEnvironment in any new applications as the others are only provided for compatibility.

The other calls are OS_Control (SWI &0F), OS_SetEnv (SWI &12), OS_CallBack (SWI &15), OS_BreakCtrl (SWI &18) and OS_UnusedSWI (SWI &19).

OS_ReadDefaultHandler allows you to get the address and details of any of the default handlers. This would be used if you wished to set up a well-defined state before running a subprogram: for example, the Desktop does so.

Details of Handlers

When a handler is called, you should not expect to be able to see the foreground application's registers. You should **only** rely on those registers explicitly defined in each handler as being meaningful on entry.

You should take care not to corrupt R14_SVC during handler code. This implies saving it on the stack if you use SWIs; see the chapter entitled *Interrupts and handling them* on page 1-109 for details. The details of each of the handlers follows:

Undefined instruction, Prefetch abort, Data abort and Address exception

These handlers are all called from hardware vectors. For a description of them see the chapter entitled *Hardware vectors* on page 1-103. These handlers are all entered with the processor in SVC mode.

All of the default handlers simply generate errors, which are passed to the current error handler.

Error

The error handler is called after any error has been generated. It is called by the default owner of the error vector; thus any routines using this vector should always 'pass it on'. Continuing after an error is not generally recommended. You should always use the X form SWIs if you wish to stay in control even when an error occurs.

The error handler is entered in User mode with interrupts enabled. Note that if the error handler is set up using OS_ChangeEnvironment, the workspace pointer is passed in R0, not R12 as is usual for other handlers.

The error buffer (the address of which should be set along with the handler address) contains the following:

Offset	Contents
0 - 3	PC when error occurred
4 - 7	Error number provided with the error.
8...	Error string, terminated with a 0

The default error handler reports the error message and number – although applications frequently set up their own error handlers. BASIC is one such example.

BreakPoint

This handler is called when the SWI OS_BreakPt (SWI &17) is called. All the user mode registers are dumped into a buffer (the address of which should be set along with the handler address) and then the handler is entered in SVC mode. You can specify a pointer to workspace to pass in R12 when this handler is called.

The following code is suitable to restore the user registers and return:

```
ADR     R14, saveblock          get address of saved registers
LDMIA   R14, {R0-R14}^          load user registers from block;
                                note that user R13,R14 are altered
LDR     R14, [R14,#15*4];       load user PC into SVC R14
MOVS    PC, R14                 return to correct address and mode
```

The default handler displays the message 'Break point at &xxxxx' and calls OS_Exit.

Escape

This handler is called when an escape condition is detected. See the chapter entitled *Character Input* on page 2-337 for details of this. You can specify a pointer to workspace to pass in R12 when this handler is called.

When the handler is entered, registers have the following values:

R11	bit 6 set, implying escape condition
R12	pointer to workspace, if set up – never contains 1
R13	a full, descending stack pointer

To continue after an escape, the handler should reload the PC with the contents of R14. If R12 contains 1 on return then the CallBack handler will be used. Typically (eg for BASIC), the handler will set an internal flag which is checked by the foreground program.

Event

This handler is called by the default owner of EventV when an event occurs. You can specify a pointer to workspace to pass in R12 when this handler is called.

When the handler is entered the processor is in either SVC or IRQ mode, with the following register values:

R0	event reason code
R1...	parameters according to event code
R12	pointer to workspace, if set up – never contains 1
R13	a full, descending stack pointer

To continue after an event, the handler should reload the PC with the contents of R14. If R12 contains 1 on return then the CallBack handler will be used.

Exit

This handler is called when the SWIs OS_Exit (SWI &11) or OS_ExitAndDie (SWI &50) are called. It is entered with the processor in user mode. You can specify a pointer to workspace to pass in R12 when this handler is called.

Unused SWI

This handler is called by the default owner of the UKSWIV. (If RISC OS can't decode the number of a SWI into one which it supports directly, it offers it as a service call to modules. If none of them claim the service, it then calls the vector UKSWIV. This allows a user routine on that vector to try to deal with the SWI. If there is no such routine, or the one(s) that is present passes the call on, then the default owner of the vector calls the Unused SWI handler.)

You can specify a pointer to workspace to pass in R12 when this handler is called.

When the handler is entered the processor is in SVC mode, with interrupts in the same state as the caller. The registers have the following values:

R11	SWI number (Bit 17 clear)
R13	SVC stack pointer
R14	user PC with V cleared

R10, R11 and R12 are stacked and are free for your own use.

UpCall

This handler is called by the default owner of UpCallV when OS_UpCall (SWI &33) is called. OS_UpCall (SWI &33) is used to warn your program of errors and situations that you may be able to recover from. See the chapters entitled *Software vectors* on page 1-59 and *Communications within* RISC OS on page 1-167. You can specify a pointer to workspace to pass in R12 when this handler is called.

CallBack

This handler is called whenever RISC OS's internal CallBack flag is set, and the system next exits to User mode with interrupts enabled. It uses a buffer (the address of which should be set along with the handler address) in which all the registers are dumped when the handler is called. You can specify a pointer to workspace to pass in R12 when this handler is called. A more detailed description follows.

Callbacks in more detail

There are two types of CallBack usage under RISC OS:

- Transient callbacks are placed in a list by calling OS_AddCallBack (SWI &54). They are used to deal with a specific case, and are called once before being removed.
- The callback handler is permanent and takes all callbacks that are not intercepted by transients. These CallBacks are explicitly requested by calling OS_SetCallBack (SWI &1B). They can also be implicitly requested by setting

R12 to 1 on exit from either an escape or event handler. There is a system default CallBack handler, but you can of course replace it using OS_ChangeEnvironment.

Transient CallBacks

Transient callbacks may be called on the system being threaded out of – that is, when it enters User mode with interrupts enabled. They can also be called when RISC OS is idling; for example, while it is waiting in OS_ReadC.

Transient callbacks are usually set up by an interrupt routine that needs to do complex processing that would take too long in an interrupt, or that needs to call a non-re-entrant SWI. OS_AddCallBack tells RISC OS that the interrupt routine wishes to be 'called back' when the machine is in a state that no longer imposes the restrictions associated with an interrupt routine. OS_RemoveCallBack removes a transient callback; this is most useful if the module is being killed before the transient callback has been serviced.

Transient CallBacks can safely be used by many clients.

Other CallBacks

The CallBack handler is only ever called on the system being threaded out of – that is, when it enters User mode with interrupts enabled. Unlike transient CallBacks, it is not called when RISC OS is idle. This means that you cannot rely on being called back within **any** given time. You **must** take this into consideration before using a CallBack handler.

Also, you **must not** allow a second CallBack before your first one has completed; see the section entitled *Application Notes* on page 1-324 for an example of how to implement a semaphore to prevent this.

The CallBack code is called in IRQ or supervisor mode with interrupts disabled. The PC stored in the save block will be a user mode PC with interrupts enabled. Note that if the currently active program has interrupts disabled or is running in supervisor mode, CallBack is not used.

In the simple case the CallBack routine should be exited by:

```
ADR     R14, saveblock          get address of saved registers
LDMIA   R14, {R0  -  R14}^      load user registers from block -
                                note that user R13,R14 are altered
LDR     R14, [R14,#15*4];       load user PC into SVC R14
MOVS    PC, R14                 return to correct address and mode
```


Currently active object pointer

This is a pointer to the address of: the last application started, or the last error handler called, or the last exit handler called. It is used by OS_Module to determine whether a module can be killed.

Setting up and restoring the environment

In order to deal correctly with the various ways in which applications can be run, and killed off, the following approach has been developed for setting up the program environment when an application starts, and restoring it when it is killed.

The basic problem is that if a new application is started 'on top' of the currently active one, it should completely replace the first, and should therefore have the same 'parent' environment as the first application. In order for this to happen, run-time language libraries and BASIC must be written so that they get themselves out of the way as a new application is started up in the same task space.

This also applies to machine-code programs which run as applications, for example modules which run as Wimp tasks.

There are two possible approaches:

- Do not set up **any** handlers **at all**, and **always** call the 'X' form of SWIs, to avoid calling the error handler. If the error handler is called, the application will be terminated, as the parent error handler will be invoked.
- Set up Error, Exit and UpCall handlers as described below, so that the program environment can be restored correctly when the program terminates. You **must** provide all three of these handlers if you use any handlers at all.

Starting an application

When you start an application, you must:

1. Check that there is sufficient memory to start up – if not, call OS_GenerateError ('Not enough application memory')
2. Set up your handlers using the SWI XOS_ChangeEnvironment; store the values returned in R1-R3 so you can later restore the old handlers.

Note that you must store the previous values not only for Exit, Error and UpCall handlers, but also for any other handlers that are set up.

If your Error handler is called

If your error handler is called and you want to call the 'external' error handler (eg BASIC if '-quit' was on the command line), you should:

1. restore all handlers to their original values (R1 - R3 for each)

2 call OS_GenerateError

If your Exit handler is called

If your exit handler is called you should:

1 restore all handlers to their original values (R1 - R3 for each)

2 call OS_Exit

If your UpCall handler is called

If your UpCall handler is called and R0 = UpCall_NewApplication (256), you should:

1 restore all handlers to their original values (R1 - R3 for each)

2 return to the caller, preserving all registers (ie carry on and start the new application)

The approach described ensures that it is not possible for the application to be terminated without it first restoring the handlers to their original values.

SWI Calls

OS_Control (SWI &0F)

Read/write handler addresses

On entry

R0 = address of error handler, or 0 to read
R1 = pointer to buffer for the error handler, or 0 to read
R2 = address of escape state change handler, or 0 to read
R3 = address of event handler, or 0 to read

On exit

R0 = previous error handler address
R1 = previous buffer address
R2 = previous escape routine address
R3 = previous event handler address

Interrupts

Interrupts are not enabled
Fast interrupts are enabled

Processor Mode

Processor is in IRQ or SVC mode

Re-entrancy

SWI cannot be re-entered as interrupts are disabled

Use

OS_Control sets some of the exception handlers. The addresses of the error handler, error handler buffer, escape state change handler and event handler are passed in R0 - R3. Zero for any of these means no change – hence you can read the current value.

Note that the call OS_ChangeEnvironment provides all of the facilities that this call provides, and should be used in preference. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_ChangeEnvironment (SWI &40)

Related vectors

ChangeEnvironmentV

OS_GetEnv (SWI &10)

Read environment parameters

On entry

—

On exit

R0 = address of the * Command string
R1 = permitted RAM limit (ie highest address available + 1)
R2 = address of the real time the program was started (5 bytes)

Interrupts

Interrupt status is unaltered
Fast interrupt status is unaltered

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI reads some information about the program environment. The value returned in R0 is the address of a copy of the command line. R1 returns the address of the byte above the last one available to the application. The five bytes pointed to by R2 give the real time: ie centiseconds since 00:00:00 01-Jan-1900.

The memory limit described in R1 can be altered by reason code 0 of OS_ChangeEnvironment.

OS_WriteEnv allows you to set these values.

Related SWIs

OS_WriteEnv (SWI &48), OS_ChangeEnvironment (SWI &40)

Related vectors

None

OS_Exit (SWI &11)

Pass control to the most recent exit handler

On entry

R0 = pointer to error block
R1 = 'ABEX' (&58454241) if return code is to be set
R2 = return code

On exit

Never returns

Interrupts

Interrupt status is unaltered
Fast interrupt status is unaltered

Processor Mode

Processor is in USR mode

Re-entrancy

SWI is not re-entrant

Use

When OS_Exit is called, control returns to the most recent exit handler address. The BASIC statement QUIT performs an OS_Exit. Before executing OS_Exit, however, you should restore any of the handlers changed in starting the application.

If the exiting program wishes to return with a result code, it must set R1 to the hex value shown above, and R2 to the desired value. Non-error results must be in the range 0 to the value of the variable SysSRCLimit. The return value is assigned to the variable SysSReturnCode, which can be interrogated by any program using OS_ReadVarVal.

To return with an error, exit with a value less than zero or greater than SysSRCLimit (having restored the previous error handler, as indicated above). This gives the error 'Return code limit exceeded' (&1E2), but still sets the variable to the required value.

Related SWIs

OS_ExitAndDie (SWI &50)

Related vectors

None

OS_SetEnv (SWI &12)

Set environment parameters

On entry

R0 = address of the handler for OS_Exit, or 0 to read
R1 = address of the end of memory limit for OS_GetEnv to read, or 0 to read
R4 = address of handler for undefined instructions, or 0 to read
R5 = address of handler for prefetch abort, or 0 to read
R6 = address of handler for data abort, or 0 to read
R7 = address of handler for address exception, or 0 to read

On exit

R0 = address of previous handler for OS_Exit
R1 = address of previous end of memory limit for OS_GetEnv to read
R4 = address of previous handler for undefined instructions
R5 = address of previous handler for prefetch abort
R6 = address of previous handler for data abort
R7 = address of previous handler for address exception

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_SetEnv sets several of the handlers for a program.

Note that the call OS_ChangeEnvironment provides all of the facilities that this call provides, and should be used in preference. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_ChangeEnvironment (SWI &40)

Related vectors

None

OS_CallBack (SWI &15)

Set up the CallBack handler

On entry

R0 = address of the register save block, or 0 to read
R1 = address of the CallBack handler, or 0 to read

On exit

R0 = address of previous register save block
R1 = address of previous CallBack handler

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_CallBack sets up the address of the CallBack handler and the register save block, zero for either value meaning no change – hence you can read the current value.

Note that the call OS_ChangeEnvironment provides all of the facilities that this call provides, and should be used in preference. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_ChangeEnvironment (SWI &40)

Related vectors

ChangeEnvironmentV

OS_BreakPt (SWI &17)

Cause a break point trap to occur and the BreakPoint handler to be entered

On entry

—

On exit

—

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

When OS_BreakPt is executed, all the user mode registers are saved in a block and the BreakPoint handler is called. The saved registers are only guaranteed to be correct for user mode.

The default handler displays the message 'Break point at &xxxx' and calls OS_Exit.

This SWI would be placed in code by the debugger at required breakpoints.

Related SWIs

OS_BreakCtrl (SWI &18)

Related vectors

None

OS_BreakCtrl (SWI &18)

Set up the BreakPoint handler

On entry

R0 = address of the register save block, or 0 to read
R1 = address of the control routine, or 0 to read

On exit

R0 = address of previous register save block
R1 = address of previous control routine
V is always clear

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_BreakCtrl sets up the address of the BreakPoint handler and the register save block, zero for either value meaning no change – hence you can read the current value.

Note that the call OS_ChangeEnvironment provides all of the facilities that this call provides, and should be used in preference. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_BreakPt (SWI &17)

Related vectors

ChangeEnvironmentV

OS_UnusedSWI (SWI &19)

Set up the handler for unused SWIs

On entry

R0 = address of the unused SWI handler, or 0 to read

On exit

R0 = address of previous unused SWI handler

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not enabled

Use

OS_UnusedSWI sets up the address of the UnusedSWI handler, zero meaning no change – hence you can read the current value.

Note that the call OS_ChangeEnvironment provides all of the facilities that this call provides, and should be used in preference. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_ChangeEnvironment (SWI &40)

Related vectors

ChangeEnvironmentV

OS_SetCallBack (SWI &1B)

Cause a call to the CallBack handler

On entry

—

On exit

—

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered because interrupts are disabled

Use

OS_SetCallBack sets the CallBack flag and so causes entry to the CallBack handler when the system next exits to user mode code with interrupts enabled (apart, of course, from the exit from this SWI). This SWI may be used if the code linked into the system (via a vector or as a SWI handler, etc) is required to do things on exit from the system.

Related SWIs

OS_CallBack (SWI &15)

Related vectors

None

OS_ReadVarVal (SWI &23)

Read a variable value

On entry

R0 = pointer to name, may be wildcarded (* and #)
R1 = pointer to buffer
R2 = maximum length of buffer
R3 = name pointer (or 0 for first call).
R4 = 3 if an expanded string is to be returned

On exit

R0, R1 preserved
R2 = number of bytes read
R3 = new name pointer, string is null-terminated
R4 = type of variable (string, number or macro)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadVarVal reads a variable and returns its value and its type. On entry, R3 should be 0 the first time the call is made for a wildcarded name, and thereafter preserved from the previous call. This enables all matches of a wildcarded name to be found. On exit, R3 points to the name of the variable found. The XOS_ReadVarVal form of the call should be used if you don't want an error to occur after the last name has been found.

You can call XOS_ReadVarVal to check for the existence of a variable by setting R2 to a value less than zero (bit 31 set) on entry. If it is still negative on exit, the variable exists; if it is zero, the variable does not exist. When using the call in this manner, you may get an error on exit, which you should ignore.

The type of the variable read is returned in R4 as follows:

Value		Type
VarType_String	(0)	String
VarType_Number	(1)	4 byte (signed) integer
VarType_Macro	(2)	Macro

R4, if set to 3 on entry, indicates that a suitable conversion to a string should be performed. String variables are unaltered, numbers are converted to (signed) decimal strings, and macros are OS_GSTrans'd.

If R4 isn't 3 on entry, the un-OS_GSTrans'd version of a macro is returned, and the four-byte binary of a number is returned.

See the section entitled *Application Notes* on page 1-324 for an example of reading a variable.

Related SWIs

OS_SetVarVal (SWI &24)

Related vectors

None

OS_SetVarVal (SWI &24)

Write a variable value

On entry

R0 = pointer to name. This can be wildcarded for update/delete
R1 = pointer to value
R2 = length of value. Negative means destroy the variable
R3 = name pointer or 0 for first call
R4 = type

On exit

R0 - R2preserved
R3 = new context pointer
R4 = type created if expression is evaluated

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_SetVarVal either creates, updates or destroys a variable. The name may be terminated by any character whose ASCII value is 32 or less and may be wildcarded if it is to be updated or deleted (ie if it already exists).

The pointer to the value to be assigned in the case of create/update is given by R1. If it is a string then it must be terminated by a linefeed (ASCII 10) or carriage return (ASCII 13) or null (ASCII 0). The interpretation of the value depends on the type given in R4 as follows:

Value	Type	
VarType_String	(0)	OS_GSTrans the given value
VarType_Number	(1)	Value is a 4 byte (signed) integer
VarType_Macro	(2)	Copy value (may be OS_GSTrans'd on use)
VarType_Expanded	(3)	The value is a string which should be evaluated as an expression using OS_EvaluateExpression, and assigned to a number or string variable, depending on the expression type
VarType_Code	(16)	Special case (see below)

If the call is successful, R3 is updated to point to the new context so allowing the next match of a wildcarded name to be obtained on a subsequent call. R4 returns the type created if an expression was evaluated (ie if R4 was 3 on entry).

R2 must be negative on entry to delete a variable. Also, to delete a type-16 variable, R4 should contain 16 on entry.

VarType_Code

When R4 is set to 16 on entry (and R2 ≥ 0) a code variable may be created. In this case R1 is the pointer to the code fragment associated with the variable, and R2 is the length of the code fragment. This code must be word-aligned and takes the following format:

Offset	Contents
0	Branch instruction to entry point for write operation
4	Entry point for read operation
8...	Body of code...

Values are always written to (and read from) code variables as strings. The entry for the write operation is called whenever the variable is to be set, as follows:

On entry

R1 = pointer to the value to be used
R2 = length of value

On exit

R1, R2, R4, R10 - R12 may be corrupted

The entry for the read operation is called whenever the variable is to be read by a call to OS_ReadVarVal, as follows:

On entry

—

On exit

R0 = pointer to value
R1 = corrupted
R2 = length of value

Both entries are called in SVC mode, therefore if any SWIs are used, R14 must be saved on the stack so that it does not become corrupted. The SVC stack is used, and no workspace is reserved. You can return errors by setting the V flag as usual.

See the section entitled *Application Notes* on page 1-324 for an example of a code variable.

Note that when a function key is input, the appropriate variable Key*Sn* is read using OS_ReadVarVal. Therefore by creating your own code variables with these names, you can cause the reading of a function key to cause a routine to be called instead of just a string being read.

Errors

OS_SetVarVal can return the following errors:

- Bad name — Wildcards/control characters in name when creating
- Bad string — OS_GSTrans unable to translate string
- Bad macro value — Control characters in the value string (R1)
- Bad expression — Expression cannot be evaluated
- Variable not found — For deletion or update
- No room for variable — Not enough room to create/update it (system heap full)
- Variable value too long — Variables are limited to 256 bytes
- Bad variable type

Related SWIs

OS_ReadVarVal (SWI &23)

Related vectors

None

OS_ChangeEnvironment (SWI &40)

Install a handler

On entry

R0 = handler number
R1 = new address, or 0 to read
R2 = R12 with which to call the routine, or 0 to read
R3 = buffer pointer if appropriate, or 0 to read

On exit

R0 preserved
R1 = previous address
R2 = previous R12
R3 = previous buffer pointer

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ChangeEnvironment is a single routine which performs the actions of OS_Control, OS_SetEnv, OS_CallBack, OS_BreakCtrl, and OS_UnusedSWI. In fact, all of those routines use this call. In new programs, you should always use this call in preference to the earlier ones.

For full details of the handlers, see the section earlier in this chapter.

On entry, R0 contains a code which determines which particular handler's address is to be set up. The new address is passed in R1. R0 also determines whether R2 and R3 are relevant or not. This is summarised in the table below:

R0	Handler	R2	R3
0	MemoryLimit	Ignored	Ignored
1	Undefined ins.	Ignored	Ignored
2	Prefetch abort	Ignored	Ignored
3	Data abort	Ignored	Ignored
4	Address exception	Ignored	Ignored
5	Other exceptions	Ignored	Ignored
6	Error	R0 when called	Error buffer address
7	CallBack	R12 when called	Register buffer address
8	BreakPoint	R12 when called	Register buffer address
9	Escape	R12 when called	Ignored
10	Event	R12 when called	Ignored
11	Exit	R12 when called	Ignored
12	Unused SWI	R12 when called	Ignored
13	Exception registers	Ignored	Ignored
14	Application space	Ignored	Ignored
15	Currently active object	Ignored	Ignored
16	UpCall	R12 when called	Ignored

The 'Memory limit' (handler 0) is the permitted RAM limit, as used by OS_GetEnv. The 'Application space' (handler 14) is the amount of read/write memory in application space. Consequently it should always be the case that Application space ≥ Memory limit.

'Other exceptions' (handler 5) is for future expansion.

Handler 13 sets the address of the area in memory where the registers are dumped when one of the exceptions (1 - 5) occurs, if the default handlers are used.

Note that in order to perform its function, OS_ChangeEnvironment vectors through ChangeEnvironmentV. A routine linked onto this vector can stop the change from happening by setting R1 (and if appropriate R2, R3) to zero and passing the call on; see the chapter entitled *Softwars vectors* on page 1-59.

Related SWIs

OS_Control (SWI &0F), OS_SetEnv (SWI &12), OS_CallBack (SWI &15)
OS_BreakCtrl (SWI &18), OS_UnusedSWI (SWI &19)

Related vectors

ChangeEnvironmentV

OS_WriteEnv (SWI &48)

Set the program environment command string and start time

On entry

R0 = pointer to environment string
R1 = pointer to start time

On exit

R0, R1 preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call sets the string that an application would read as its command string, containing parameters for the application. This SWI also sets the start time, which is the real-time, stored as a 5 byte value.

This SWI is mainly used for debuggers.

Related SWIs

OS_GetEnv (SWI &10)

Related vectors

None

OS_ExitAndDie (SWI &50)

Pass control to the most recent exit handler and kill a module

On entry

R0 = pointer to error block
R1 = 'ABEX' (&58454241) if return code is to be set
R2 = return code
R3 = pointer to module name

On exit

never returns

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI is like OS_Exit, except that it will kill a module before exiting. R3 points to a string containing the module's name.

Related SWIs

OS_Exit (SWI &11)

Related vectors

None

OS_AddCallBack (SWI &54)

Add a transient callback to the list

On entry

R0 = address to call
R1 = value of R12 to be called with

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

A transient callback is placed on a list of tasks who want to be called as soon as RISC OS is not busy. Usually, this will be just before returning from a SWI or while waiting for a key and so on.

This SWI will place a transient routine on that list. It is usually called from an interrupt routine that needs to do complex processing that would take too long in an interrupt, or that needs to call a non-re-entrant SWI. Note that it is not necessary to call OS_SetCallBack. Using this SWI means you want to be called. OS_SetCallBack is only needed when using the callback handler.

A routine called by this mechanism must preserve all registers and return by

```
MOV PC, R14
```

Related SWIs

None

Related vectors

None

OS_ReadDefaultHandler (SWI &55)

Get the address of the default handler

On entry

R0 = reason code (0 - 16)

On exit

R0 preserved
R1 = address of default handler
R2 = workspace address
R3 = buffer address

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

Using the same handler number in R0 as those in OS_ChangeEnvironment (see page 1-308), this SWI returns details about the default handler.

Zero in R1, R2 or R3 on exit means that it is not relevant.

Related SWIs

OS_ChangeEnvironment (SWI &40)

Related vectors

None

OS_RemoveCallBack (SWI &5F)

Removes a transient callback from the list

On entry

R0 = address that was to be called
R1 = value of R12 that the routine was to be called with

On exit

R0, R1 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call removes a transient callback from the list. You should do so if your module has an outstanding callback request, but will not be able to service the request when it is granted – for example if the module is being killed.

This call is not available in RISC OS 2.0, which can cause problems. For example, if a module is being killed and it has outstanding callback requests, it must refuse to die, otherwise the callback may be granted after that memory has been reused for something else.

Related SWIs

None

Related vectors

None

*Commands

*Go

Calls machine code at the given address

Syntax

`*Go [hexadecimal_address] [ ; environment]`

Parameters

hexadecimal_address	address of machine code to call
environment	environment string to pass to machine code

Use

*Go calls machine code at the given address, passing it an optional environment string. If the address is omitted, it defaults to &8000, which is where application programs (such as the C compiler) are loaded.

*Go enters an application, and you cannot use it to run machine code subroutines.

Example

`*Go 9000 ; SrcList`	Call machine code at &9000, passing it the string 'SrcList'

Related commands

None

Related SWIs

None

Related vectors

None

*Quit

Exits from the current application

Syntax

```
*Quit
```

Parameters

None

Use

*Quit exits from the current application – that is, it returns to the previous context.

Related commands

*OS

Related SWIs

None

Related vectors

None

*Set

Assigns a string value to a system variable

Syntax

```
*Set varname value
```

Parameters

varname	a variable name, or a wildcard specification for a single variable name
value	this parameter depends on the system variable referred to in the *varname* specification, and is GSTrans'd before use

Use

*Set assigns a string value to a system variable, like an assignment statement in a programming language. For example:

```
*Set varname text
```

assigns the string 'text' to the variable varname.

Aliases

Another use for the *Set command is to change the name of a command to one which is more convenient for the user:

```
*Set Alias$name cname
```

establishes *name* as an alternative name for the command *cname*; for example after:

```
*Set Alias$Aid Help
```

the command *Aid is now a synonym for *Help; both commands access the help system. Another example is:

```
*Set Alias$Mode Echo |<22>|<%0>
*Mode 12
```

The command implements a new command *Mode, which sets the screen to mode 12 (in the above case). The Echo command reflects the string which follows it; |<22> generates the ASCII character 22, Ctrl V, which is equivalent to the VDU command to change mode. |<%0> reads the first parameter from the command line, and generates the corresponding ASCII code.

The command `*Show Alias$*` lists all aliases.

Example

```
*Set Sys$Year 1988
```

Related commands

*SetEval, *SetMacro, *Unset

Related SWIs

OS_GSTrans (SWI &27)

Related vectors

None

*SetEval

Evaluates an expression and assigns its value to a system variable

Syntax

`*SetEval` *`varname expression`*

Parameters

`varname`	a valid variable name
`expression`	a valid Command Line expression

Use

*SetEval evaluates an expression and assigns its value to a system variable.

See the section entitled ***Evaluation operators*** on page 1-432 for a description of the operators that you can use.

Example

```
*Set rate 12
*SetEval rate rate + 1
*Show rate
rate (Number) : 13
*SetEval fred "jim"+"sheila"
```

Related commands

*Set, *SetMacro, *Unset, *Eval

Related SWIs

None

Related vectors

None

*SetMacro

Assigns an expression to a system variable

Syntax

`*SetMacro varname expression`

Parameters

varname	a valid variable name
expression	a valid Command Line expression

Use

*SetMacro assigns an expression to a system variable. The parameters making up the expression are not interpreted when the command is given, but each time the variable is used.

See the section entitled *Evaluation operators* on page 1-432 for a description of the operators that you can use.

Example

```
*SetMacro CLI$Prompt "<Sys$Time> "
13:43:17                    system time replaces existing prompt
Return                      Return key pressed two seconds later
13:43:19                    new system time displayed as prompt
```

This resets the Command Line prompt, which appears as the first item on each line, to be the current time whenever the prompt is given. Compare this with using the *Set command:

```
*Set fred <Sys$Time>
*Show fred
FRED  :  13:43:59
```

the *Show command issued five minutes later will produce:

```
*Show fred
FRED  :  13:43:59
```

Notice that the time is fixed at the time the *Set command is last used, in contrast to the *SetMacro command.

Related commands

*Set, *SetEval, *Unset

Related SWIs

None

Related vectors

None

*Show

Displays the list of system variables

Syntax

```
*Show [variable_spec]
```

Parameters

variable_spec	a variable name or a wildcard specification for a set of variable names

Use

*Show displays the name, type and current value of any system variables matching the name given as a parameter. These include the 'special' system variables, which may be altered, but which cannot be deleted.

If no name is given, all system variables are displayed.

Example

```
*Show                    lists all system variables
*Show CLI$Prompt
*Show Alias$*            lists all aliases
```

Related commands

*Set, *SetEval, *SetMacro

Related SWIs

None

Related vectors

None

*Unset

Deletes a system variable

Syntax

```
*Unset variable_spec
```

Parameters

variable_spec	a variable name or a wildcard specification for a variable name

Use

*Unset deletes a system variable, which may be specified using wildcards.

Example

```
*Unset My_var
```

Related commands

*Set, *SetEval, *SetMacro

Related SWIs

None

Related vectors

None

Application Notes

Reading a variable

Here is a short example of reading a variable using OS_ReadVarVal:

```
;Print all sys$ variable names
        ADR     R1, valBuffer           ;Buffer to place value
        MOV     R3, #0                  ;Initial context
.loop
        ADR     R0, strName             ;Wildcarded name to find
        MOV     R2, #bufferLen          ;Length of value buffer
        SWI     "XOS_ReadVarVal"        ;Non-error reporting one
        MOVVSS  PC,R14                  ;Return and clear V
        MOV     R0, R3                  ;Get address of name
        SWI     "OS_Write0"             ;Print it
        SWI     "OS_NewLine"            ;and new line
        B       loop                    ;again
        ....
.strName        EQUS    "SYS$*" + CHR$0
```

Checking the value of a variable

The short code fragment below checks if a variable has a particular value, without giving an error if it does not exist or contains quotes.

```
*SetMacro App$Temp <Variable>
If App$Temp = "desired_value" Then commands.
```

Don't forget to `Unset` the `App$Temp` macro when you have finished using it.

Code variable

Below is a complete example of a program to create a variable called Mode. The read action is to return the current display mode, and the write action to set the mode.

```
.start  ADR     R0, varName             ;Pointer to the name
        ADR     R1, code                ;Start of code body
        MOV     R2, #endCode-code       ;Length of code body
        MOV     R3, #0                  ;Context pointer
        MOV     R4, #&10                ;'special' type
        SWI     "OS_SetVarVal"          ;Create it
        MOV     PC, R14                 ;Return

.code
        B       writeCode               ;Branch to write code
.readCode
        STMFD   R13!, {R14}             ;Save return address
        MOV     R0, #&87                ;OS_Byte read mode number
```



```
        SWI     "XOS_Byte"
        MOV     R0, R2                  ;Mode in R0 for conversion
        ADR     R1, buffer              ;Buffer for ASCII conversion
        MOV     R2, #4                  ;Max len of buffer
        SWI     "XOS_BinaryToDecimal"
        MOV     R0, R1                  ;Pointer in R0
                                        ;length already in R2
        LDMFD   R13!, {PC}              ;Return

.writeCode
        STMFD   R13!, {R14}             ;Save return address
        SWI     "XOS_ReadUnsigned"      ;R1 set correctly already
        SWIVC   &20100+22               ;VDU mode change
        MOVVC   R0,R2                   ;Get integer read in R0
        SWIVC   "XOS_WriteC"            ;Do mode change
        LDMFD   R13!, {PC}              ;Return

.buffer
        EQUD    0                       ;Buffer for string conversion
.endCode

.varName
        EQUS    "Mode "                 ;Name of variable
```

The routine at 'start' creates the variable. Obviously as the code body is copied into the system heap, it must be position independent. The two routines readCode and writeCode are called whenever an access to the variable is made. For example, a *Set Mode command will call the write code entry, and *Show SysSMode or *Echo <Mode> will call the read entry.

Notice that in the body of the code variable, only XOS_ SWIs are used. This is because it is important that errors are not generated when the read or write code executes. A more rigorous version of the code above would check V after each SWI and return if it was set.

OS_AddCallBack

The next example shows the use of OS_AddCallBack; it prints 'Run away!' after 2 seconds:

```
DIM code 100
P%=code
[
.alarm STMFD r13!, {r14}
       SWI "XOS_WriteS"
       EQUS "Run away!"
       EQUB 10:EQUB 13:EQUB 0
       ALIGN
       LDMFD r13!, {r15}
.timer  STMFD r13!, {r0,r14}
       MOV r0, r12                      ; set up for us by BASIC bit
                                        ; r12 is not used in alarm,
                                        ; so r1 here is don't-care
       SWI "XOS_AddCallBack"
       LDMFD r13!, {r0, r15}
]
SYS "OS_CallAfter",200,timer,alarm
```

A CallBack handler

The final example shows a callback handler, with a semaphore to prevent recursive callback; it prints 'Run away!' when mouse buttons are pressed.

```
DIM code 200
P%=code
[
.sema   EQUB 1
       ALIGN
.saveblock :]:P%=P%+16*4:[
.callback                               ; entered here in a privileged mode,
                                        ; with interrupts disabled
                                        ; first thing to do is enable IRQs
       TEQP r15, #3                     ; force SVC mode, IRQs on.
       SWI "XOS_WriteS"
       EQUS    "Run away!"
       EQUB 10:EQUB 13:EQUB 0
       ALIGN
       ADR r14, saveblock
       LDMIA r14, {r0-r14}^             ; most registers reloaded
       TEQP r15, #3+(1<<27)             ; disable IRQs for sema update
       MOVNV r0, r0                     ; and return
       MOV r14, #1
       STRB r14, sema
       LDR  r14, saveblock+15*4         ; must not allow another callback
                                        ; request until the stashed PC is safe
       MOVS r15, r14                    ; return, enabling IRQs etc
.events
       CMP r0, #10                      ; mouse button state change?
       MOVNES r15, r14                  ; no - run away
       STMFD r13!, {r14}
       LDRB r12, sema                   ; possibly request callback
       MOV r14, #0
       STRB r14, sema                   ; and disable any futher requests
       LDMFD r13!, {r15}                ; until that one serviced.
]
SYS"OS_CallBack",saveblock,callback,0 TO osave,ocall
REM Note that we aren't using r12 in the callback handler;
REM if this was in a module, for example, sema would be in the workspace,
REM and we would have to access it r12-relative; r12 would therefore be
REM set to be the workspace pointer on entry.
SYS "OS_ChangeEnvironment",10,events,0 TO ,oldev
*FX 14,10
REPEAT UNTILINKEY -1: REM loop until shift
*FX 13,10
SYS "OS_ChangeEnvironment",10,oldev,0
SYS "OS_CallBack",osave,ocall,0
REM Note that in both the above calls, the R12 values are explicitly left
REM alone, because we didn't use them earlier.
```

16 Memory Management

Introduction

This chapter describes the memory management in RISC OS. This covers memory allocation by a program or module as well as using the MEMC chip to handle how memory is mapped.

In many environments, such as BASIC and C, you can use the language's intrinsic memory allocation routines, which use the calls described in this chapter transparently. For example, refer to Wimp_SlotSize on page 4-270 of the chapter entitled *The Window Manager*.

Similarly, small, transiently loaded utilities may not require any memory over the 1024 bytes they are automatically allocated. Some programs and modules, however, will require arbitrary amounts of memory, which can be freed after use. For example, filing systems, specialised VDU drivers such as the font manager and so on. The memory manager provides simple allocation and deallocation facilities. Relocatable modules can use this manager either directly, to manipulate their own private heap, or indirectly using the module support calls.

A block of memory can be set up as a heap. This is a structure that allows arbitrary parts of the block to be allocated and freed. A program simply requests a block of a given size and is given a pointer to it by the heap manager. This block can be expanded or contracted or freed by using this pointer as a reference.

The part of the screen RAM that is not visible on the screen is also available as a temporary buffer. This memory is temporarily available because of the way that vertical scrolling is done.

One of the other memory resources available is the battery-backed CMOS RAM. This is used to hold default system parameters while the power is off. Modules, applications and users may use spare locations in CMOS RAM for their own purposes.

The MEMC chip controls how logical addresses (those used by programs or modules) are mapped into the physical memory location to use. Numerous calls are used to control how it does this, though generally this is something that most programs would not want to do.

Overview

Heap manager

RISC OS contains a heap management system. This is used by the operating system to allocate space within the relocatable module area and also to maintain the system heap. A heap is just an area of memory from which bytes may be allocated, then deallocated for later use. An area can also be reallocated, meaning that its size changes.

The heap manager is also available to the user. You provide an area of memory which is to be used for the heap, which can be any size you require. If you are a module, then the heap would be a block within the RMA, and if you are a program, then it would be within the application space.

Thus, it would be a heap within a heap; for example a block in the RMA would be allocated by a module, and then declared as a heap. In theory, this process could continue indefinitely, but in practice this is as far as you need to go.

At the start of a heap, the heap manager sets up the heap descriptor, which is a block containing information on the limits of the heap, etc. This descriptor is updated by the heap manager when necessary.

When a block within this heap is required, a request is made to the heap manager, which returns a pointer to a suitable block of memory. The heap manager keeps a record of the total amount of memory which is free in the heap and the largest individual block which is available.

Heap fragmentation

The heap management system does not provide garbage collection. This is the technique of moving blocks of allocated memory around so as to maximise the contiguous free space and avoiding excessive fragmentation of the heap.

Also, the heap management system will never attempt to move a block within the heap, since it has no knowledge of whether the block contains pointers that need to be relocated, or whether there are any pointers to the block which need updating. Hence, unless an area of contiguous free space of the size requested is available, a request for a block will fail.

MEMC control

MEMC maps logical onto physical addresses. To do this, it maintains a table of 128 entries that map a given memory block to a particular address. Generally, the system will take care of the operation of this mapping for you. Calls are provided to allow you to read this mapping and alter it, but you should have a very good reason to do so, and be certain of what you are doing.

Screen memory

The vertical scrolling technique used under RISC OS is to change the memory location that the screen starts at. This means that part of the screen memory may be unused, depending on the screen mode and the amount of memory reserved. You can use this memory temporarily, as long as you don't cause any output that may scroll the screen. Also remember that this memory is limited to one program using it at a time, so it may not be available every time you request it. Consequently, you cannot count on it being there when writing a module or application.

Battery-backed CMOS RAM

A block of 240 bytes of battery-backed CMOS RAM is available under RISC OS. Each location has a specific meaning and should not be directly modified unless you are sure of the meaning of the value. Many of these locations are changed indirectly using the *Configure commands. These can be found throughout this manual, in the chapter appropriate to their function.

Some bytes are not allocated, and are reserved for users and applications to use. If you want to use one or more of the application bytes, you should request a location in writing from Acorn Computers. This is so that different applications don't accidentally use the same location.

Technical Details

Guidelines on using memory efficiently

This section provides basic information on memory management by RISC OS applications. It is intended to provide some specialist knowledge to help you write efficient programs for RISC OS, and to provide some practical hints and tips.

All the information in this chapter relating to programs written in C refers to Acorn's Desktop C product.

You should follow the guidelines in this section to make the best use of available memory. The guidelines are explained in more detail on the following pages.

- **Use recovery procedures** – Your program should keep the machine operational. Don't allow your program to lock up when memory runs out; your program should indicate that it has run out of memory (with an error or warning message) and only stop subsequent actions that use more memory. Ideally, ensure that actions which free up memory have enough reserved memory to run in.
- **Return unwanted memory** – You should return any memory you have no further use for. Claiming memory then not returning it can tie up memory unnecessarily until the machine is re-booted. RISC OS has no garbage collection, so once you have asked for memory RISC OS assumes that you want it until you explicitly return it, even if your program terminates execution. Language libraries often provide you with protection from this, as long as memory is claimed from them.
- **Don't waste memory** – You should avoid wasting memory. It is a finite resource, often wasted in two ways:
  - by permanently claiming memory for infrequent operations
  - by fragmenting it, so that although there is enough unused memory, it is either in the wrong place, or it is not in large enough blocks to use.

Recovery from lack of memory

An important consideration when designing programs for RISC OS is the recovery process, not just from user errors, but also from lack of system resources.

An example of a technique that can be designed into an application is to make an algorithm more disc-based and less RAM-based on detection of lack of memory. This could allow you to continue using an application on a small machine (especially one with a hard disc) at the expense of some speed.

When implementing your code, expect the unexpected and program defensively. Be sure that when the system resources you need (memory, windows, files etc) are not available, your program can cope. Make sure that, when a document managed by your program expands and memory runs out, the document is still valid and can be saved. Don't just check that your main document expansion routines work; check that **all** routines which require memory (or in fact any system resource) fail gracefully when there is no more.

Centralising access to system resources can help: write your program as if every operating system interface is likely to return an error.

Avoiding permanent loss of memory

Permanent loss of memory is mainly a problem for applications or modules written entirely in assembly language. When interworking assembler routines with C or another high level language you should use memory handed to you by the high level language library (eg use `malloc` to get a memory area from C and pass a pointer to it as an argument to your assembler routine). The language library automatically returns such areas to RISC OS on program exit. Additional types of program requiring care to avoid memory loss are those expected to run for a long time (eg a printer spooler) and those making use of RMA directly through SWI calls.

When using the RMA for storage directly through SWI calls, especially for items in linked lists, consider using the first word as a check word containing four characters of text to identify it as belonging to your program. When a block of RMA is deallocated, the heap manager puts it back into a list of free blocks, and in so doing overwrites the first word of the block.

This technique therefore serves two purposes:

1. after your program has been run and exited, your check word can be searched for, showing up any blocks you have failed to deallocate
2. it avoids problems when accidentally referencing deallocated memory.

A typical problem of referencing deallocated blocks results from using the first word as a pointer to your program's next block, then accidentally referencing a wild pointer when it is overwritten.

You can use the following BASIC routine to search for any lost blocks:

```
100 REM > LostMemory checks for un-released blocks
110 RMA%=&01800000: RMAEnd% = RMA% + (RMA%!12)
120 FOR PossibleBlock% = RMA%+20 TO RMAEnd%-12 STEP 16
130    REM Now loop looking for "Prog"
140    IF PossibleBlock%!0 = &676F7250 THEN
150       PRINT "Block found at &";~PossibleBlock%
160    ENDIF
170 NEXT PossibleBlock%
180 END
```

When writing relocatable module initialisation code you should check that memory and other system resources are returned if initialisation is unable to complete and is going to return with V set. It is often useful to construct module finalisation code as a mirror image of initialisation code so that it can be jumped to when initialisation is going to return an error and cleaned up. A typical algorithm is:

Initialisation

Claim main workspace: If error then keep this error and goto Exit3
Claim secondary workspace: If error then keep this error and goto Exit2
Claim tertiary workspace: If error then keep this error and goto Exit1
Return

Finalisation

Set kept error to null
Release tertiary workspace
Exit1 Release secondary workspace
Exit2 Release main workspace
Exit3 Get kept error (if there was one)
Return

Avoiding memory wastage

The key factor in writing programs that use memory efficiently and don't waste it is understanding the following:

- how SWI XOS_Module and SWI XOS_Heap work if you are constructing a relocatable module or are using the RMA from an application
- how C flex and `malloc` work when writing a C program (parts of which may be written in assembler).

This understanding will lead you to writing programs that will work in harmony with the storage allocator. See the following section for a description of C memory allocation.

The C storage manager

Understanding the C storage manager is obviously useful to writers of C. But it may also be useful to writers of assembly language for two reasons: to assist in constructing part C and part assembler programs; to assist in constructing their own memory allocation routines, both as an example algorithm and as an allocator that may be running for other applications at the same time as their own.

Normal C applications (ie those not running as modules) claim memory blocks in two main ways:

- from `malloc`
- from `flex`.

The `malloc` heap storage manager is the standard interface from which to claim small areas of memory. It is tuned to give good performance to the widest variety of programs.

In the following sections, the word *heap* refers to the section of memory currently under the control of the storage manager (usually referred to as `malloc`, or the `malloc` heap).

The flex facility is available as part of RISC_OSLib, and can be useful for claiming large areas of data space. It manages a shifting set of areas, so its operation can be slow, and address-dependent data cannot be stored in it. However, it has the following advantages:

- it doesn't waste memory by fragmenting free space
- it returns deallocated memory to RISC OS for use by other applications.

Allocation of malloc blocks

All block sizes allocated are in bytes and are rounded up to a multiple of four bytes. All blocks returned to the user are word-aligned. All blocks have an overhead of eight bytes (two words). One word is used to hold the block's length and status, the other contains a guard constant which is used to detect heap corruptions. The guard word may not be present in future releases of the ANSI C library. When the stack needs to be extended, blocks are allocated from the `malloc` heap.

When an allocation request is received by the storage manager, it is categorised into one of three sizes of blocks

- small 0 → 64
- medium 65 → 512
- large 513 → 16777216.

The storage manager keeps track of the free sections of the heap in two ways. The medium and large sized blocks are chained together into a linked list (overflow list) and small blocks of the same size are chained together into linked lists (bins). The overflow list is ordered by ascending block address, while the bins have the most recently freed block at the start of the list.

When a small block is requested, the bin which contains the blocks of the required size is checked, and, if the bin is not empty, the first block in the list is returned to the user. If there was not a block of the exact size available, the bin containing blocks of the next size up is checked, and so on until a block is found. If a block is not found in the bins, the last block (highest address) on the overflow list is taken. If the block is large enough to be split into two blocks, and the remainder is a usable size (> 12 including the overhead) then the block is split, the top section returned to the user and the remainder, depending on its size, is either put in the relevant bin at the front of the list or left in the overflow list.

When a medium block is requested, the search ignores the bins and starts with the overflow list. This is searched in reverse order for a block of usable size, in the same way as for small blocks.

When a large block is requested, the overflow list is searched in increasing address order, and the first block in the list which is large enough is taken. If the block is large enough to be split into two blocks, and the size of the remainder is larger than a small block (> 64) then the block is split, the top section is returned to the overflow list, and bottom section given to the user.

Should there not be a block of the right size available, the C storage manager has two options:

1. Take all the free blocks on the heap and join adjacent free blocks together (coalescing) in the hope that a block of the right size will be created which can then be used
2. Ask the operating system for more heap, put the block returned in the overflow list, and try again.

The heap will only be coalesced if there is at least enough free memory in it to make it worthwhile (ie four times the size of the requested block, and at least one sixth of the total heap size) or if the request for more heap was denied. Coalescing causes the following:

- the bins and overflow list are emptied;
- the heap is scanned;
- adjacent free blocks are merged;
- the free blocks are scattered into the bins and overflow list in increasing address order.

Deallocation of malloc blocks

When a block is freed, if it will fit in a bin then it is put at the start of the relevant bin list, otherwise it is just marked as being free and effectively taken out of the heap until the next coalesce phase, when it will be put in the overflow list. This is done because the overflow list is in ascending block address order, and it would have to be scanned to be able to insert the freed block at the correct position. Fragmentation is also reduced if the block is not reusable until after the next coalesce phase. It is worth noting that deallocating a block and then reallocating a block of the same size can not be relied upon to deliver the original block.

Reallocation of malloc blocks

You should be cautious when using `realloc`. Reallocating a block to a larger size will usually require another block of memory to be used and the data to be copied into it. This means that you cannot use the whole of the heap as both blocks need to be present at the same time.

If consecutive calls keep increasing the block size until all memory is used up, then only about a third of the heap is likely to be available in one block. A typical course of events is:

1. The first block is present (block A).
2. It is extended to a larger sized block (block B). Block A must still be present (see above).
3. It is again extended to a larger sized block (block C). Block B must still be present (see above). However, block A also still exists because it is too small to use, and cannot be coalesced with another block because block B is in the way.

Wimp slots and the C flex system

A typical C application running under the Wimp has a single contiguous application area (wimp slot) into which are placed the following:

- program image
- stack
- static data
- `malloc` data.

The initial wimp slot size is set by the size of the Next slot (in the Task display window) when the application is started, or by *WimpSlot commands in the !Run file associated with the C application. If the `malloc` heap is full and the operating system has free memory, the wimp slot grows, raising its highest address. Once enlarged by `malloc`, the wimp slot never reduces again until program termination.

The application area is used as follows:

low memory: the application image
the static data
high memory: the `malloc` heap

The stack is allocated on the heap, in 4K (or as big as needed) chunks: the ARM procedure call standard means that disjoint extension of the stack is possible. The only other use that the ANSI library makes of the `malloc` heap is in allocating file buffers, but even this usage can be prevented by making the appropriate calls to the ANSI library buffer handling facilities (`setvbuf`). The operation of the `malloc` heap is described above and is designed to provide good performance under heavy use. Its design is such that small blocks can be allocated and freed rapidly.

Any `malloc` heap tends to fragment over time. This is particularly serious in the following circumstances:

- no virtual memory
- multitasking – if memory is not in use, it should be handed to other applications
- if a program runs out of memory it must not crash, but must recover and continue.

These are just the conditions under which a desktop application operates!

Because of this, the flex facilities are available as part of RISC_OSLib (the RISC OS-specific C library provided with Desktop C). These provide a shifting heap, intended for the allocation of large blocks of memory which might otherwise destroy the structure of a `malloc`-style heap.

Flex works by increasing the size of the application area, using space above that reserved for use by `malloc`. When the `malloc` heap grows, flex areas are shifted. The benefits of using flex can be seen in Draw, Paint and Edit, which are all written in C using early versions of RISC_OSLib. Their application areas expand when new files are added, contract when files are discarded, and do not suffer from needless incremental application area growth over time.

The implementation of flex is quite simple. There is no free list as memory is shifted whenever a block is destroyed or changed in size. New blocks are always allocated at the top. When blocks are deallocated or resized, those above are moved. This means that deallocating or changing the size of a block can take quite a long time (proportional to the sum of the sizes of the blocks above it in memory). Flex is also not recommended for allocation of small blocks. Its other limitation is that as flex blocks can be shifted, you should not use them for address-dependent data (eg pointers or indirected icon data).

In addition to the facilities described above, RISC_OSLib also provides an obsolete `malloc`-like allocator of non-shifting blocks called heap.

Two facilities are provided, because no one storage manager can solve all problems in the absence of Virtual Memory. A program which works adequately with `malloc` should feel no compulsion to use anything else. The use of flex, however, particularly in desktop applications such as editors (which are likely to be resident on the desktop for a long period of time) can go a long way towards improving their memory usage.

Using memory from relocatable modules

Relocatable modules should use memory from three sources: the supervisor stack; the RMA; and application workspace. Use of pc-relative written data should be avoided as it makes a module unsuitable to ROM, unsuitable for multiple instantiation, and permanently reserves space, possibly only for occasional use.

The supervisor stack is small and not extendable, so care must be taken to use this resource very economically.

The RMA is the standard source of workspace for any of the non-user mode routines contained in a module, as described in the RISC OS *Programmer's Reference Manual*. Care must be taken to deallocate unwanted blocks – the marker word hint described earlier in this chapter may be useful. C malloc uses RMA when called from non-user mode.

Application workspace only belongs to a module when referenced from module user mode code running as the sole current application (with RISC OS desktop multitasking halted) or when running as a RISC OS application having dealt with the `Service_Memory (&11)` service call (sent round by the wimp when your program issues `SWI Wimp_Initialise`) to keep application workspace.

Never access your application's workspace from an interrupt routine. During interrupts, the state of the application area is effectively random. Since your interrupt routine could execute at any time, it could happen while some other application is switched in. If this did happen, and the interrupt routine updated application space, then some other application could be affected. To get around this problem, allocate some RMA space for your interrupt routine to use when it needs to; this memory will be visible when your application is running. Remember to free up the RMA space when you've finished with it.

Using memory from relocatable modules written in C

There are additional points you should note if you are writing modules in C (although most of the points made above apply equally well – particularly the preceding paragraph).

All memory allocated by `malloc` comes from the RMA when your program is executing in non-user mode. So remember to free it up when you've finished with it. If your module allocates any RMA blocks by calling SWI XOS_Module directly, the C run-time system does not clear them out when your module finalises, so make sure you do!

There are two sets of `atexit()` routines, the ones which you registered during initialisation ie before your module was entered via the `main()` entry point (because the module was RMRun for instance), and the ones you registered after. The ones registered before will be executed when your module is finalised – this is how to clear up after yourself; the ones after will be called when your module exits from being run, ie when `main()` terminates.

When you are writing a C module, use `exit()`, not SWI XOS_Exit.

When executing as C module SVC mode code (during initialisation, finalisation, service or interrupt entry) your stack will be small. Also, your stack, unlike when in USR mode (ie running as an application) will not extend dynamically. It is therefore very important to be extremely economical with stack space; avoiding large auto arrays, using `malloc` where larger spaces are required, then freeing at the routine end.

Static variables (and arrays etc.) in a C module are extant for the lifetime of the module, ie the entire time it is loaded. If they are only needed when it is running as an application, then they should be claimed using `malloc` instead.

Heap Manager

The heap is controlled by a single SWI, OS_Heap (SWI &1D). This has a reason code and can perform the following operations:

Reason code	Meaning
0	Initialise heap
1	Describe heap
2	Allocate a block from a heap
3	Free a block
4	Change the size of a block
5	Change the size of a heap
6	Read the size of a block

Internal format of the heap

A description of the structure used by the heap manager is given below. It should be noted that this structure is not guaranteed to be preserved between releases of the software and should not be relied upon. It is given purely for advanced programmers who may want to interpret the current state of the heap when testing and debugging their own code.

The heap descriptor is a block of four words:

&00	Special heap word
&04	Free list offset
&08	Heap base offset
&0C	Heap end offset

Figure 16.1 Format of heap descriptor

The 'special' heap word contains a pattern which distinguishes correct heap descriptors. The pattern is made up of the characters 'Heap' – which is &70616548 in hex.

All other words are offsets into the heap. This means that the heap is relocatable unless you place non-relocatable information in it.

The free list offset is an offset to the first free block in the heap, or zero if there are no free blocks. If the word is non-zero, the first free block is at address:

heap start + free list offset + 4

The other entries are offsets from the start of the heap which refer to boundaries within the heap structure. The heap is delimited as follows:

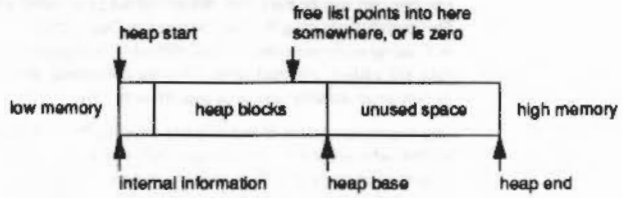


Figure 16.2 How the heap is delimited

Blocks in the free list have information in the first two words as follows:

- Word 0 is the link to the next free block or 0 if at the end

- Word 1 is the size of this block (including these two words)

Allocated blocks start with a word which holds the size of the allocated block. The pointer returned by SWI OS_Heap when a block is allocated actually points to the second word which is the start of the memory available.

Allocation forces the block size to be a multiple of eight, to ensure that no matter what you do, the fragments can always be freed. Therefore, the minimum size of area that can be initialised is 24 bytes (16 for the fixed information and 8 for a block).

Logical memory map

The organisation of the logical address space is **currently** as follows:

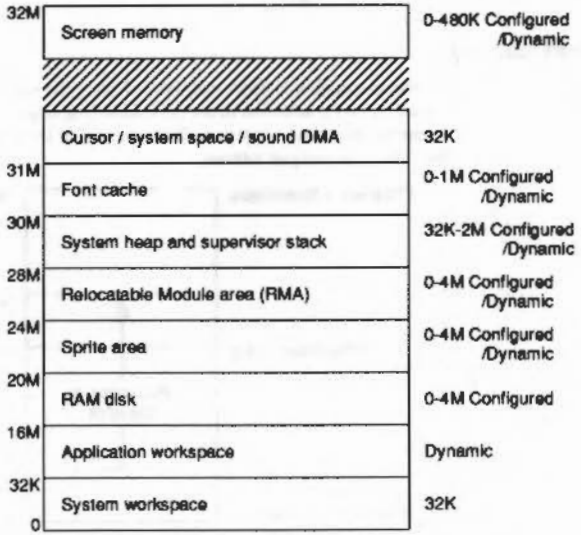


Figure 16.3 Typical logical memory map

You must not assume that any of the above addresses will remain fixed (save for the base of application workspace). There are defined calls to read any addresses you need, and you must use them.

Setting up the memory map

The memory map is set up on hard reset as follows:

- The permanent 32K allocations for system workspace at addresses &0000000 and &1F00000 (31Mbytes) are made, as well as some other fixed allocations (such as an initial part of the system heap).
- Then space is allocated to the various adjustable size regions, such as the screen., the system heap, the RMA, etc. Some of these have an absolute configured size, such as the screen. This is allocated in full. For other regions (such as the system heap and RMA), the configured size is the amount of free space that will be left; these only have a minimal allocation made at this stage.
- The rest of memory is then allocated to the application workspace, from address &8000 up.
- System ROM and expansion card modules are then initialised.
- Finally, the regions that have a configured free space get allocated. First they are shrunk as far as possible (to ensure as close to 0 bytes free as possible), then a block of the configured size is requested and freed, so that the heaps contain as close to the configured free space as possible.

Example memory allocation

Here is an example of how memory might be allocated given some typical RAM size allocations on an A310 (8K page size):

Area	Pages	Page size	Total
FontSize	20	4K	80K
RamFsSize	0	8K	0
RMASize	16	8K	128K
ScreenSize	20	8K	160K
SpriteSize	10	8K	80K
SystemSize	4	8K	32K+32K
System workspace			32K
Cursor etc. workspace			32K
Total			576K
Application area	1024K – 576K = 448K		

A configured screen size of 0 means 'default for this machine', which is 160K on an A310 (see *Configure ScreenSize).

As outlined above, the size of the system area (at 28M) is shrunk as far as possible after all module initialisation and then 'n' extra pages are added. 8K of this is used for the system stack. The rest is for OS variable storage (eg alias variables) and module information. The configured amount is added to the 32K initially allocated.

Altering the memory map

While no application is running (ie in the supervisor prompt), the memory map can be altered as required. For example, if you load a module from disc and the RMA isn't big enough to hold it, the size of the RMA will be increased by an appropriate amount. The OS can only do this when there is no application active, as the extra memory has to be taken from the application workspace. Most programs don't react too kindly to large areas of their memory allocation disappearing.

Under an environment such as the Wimp desktop, multiple applications are run concurrently. The currently running application is mapped into &8000. When the Wimp decides to swap to another application, it maps the current one out and maps the new application into that space. Thus, every application is given the illusion that it is the only one in the system. Before each call your application makes to Wimp_Poll (which is when it may be swapped out), it must call OS_DelinkApplication (SWI &4D) to remove any vectors that point into the application area – if it has any to remove, that is. When its call to Wimp_Poll returns (and hence it is swapped back in), it must then call OS_RelinkApplication (SWI &4E) to reload these vectors.

Page size

The SWI OS_ReadMemMapInfo (SWI &51) returns the pagesize used in the system and the number of pages present. For more details of page sizes, see the section entitled *Page size* on page 1-17.

Controlling memory allocation

OS_ChangeDynamicArea (SWI &2A) allows control of the space allocated to the system heap, RMA, screen, sprite area, font cache and RAM filing system. Any space left over is the application space by default. Any of these settings can be read with OS_ReadDynamicArea (SWI &5C). OS_ReadRAMFsLimits (SWI &4A) will read the range of bytes used by the RAM filing system. The size of it can be set in CMOS RAM using *Configure RamFsSize. See also *Configure RMASize and *Configure SystemSize.

Memory protection

You have read/write access to much of the logically mapped RAM. There are exceptions, such as the 32K system workspace at &1F00000 (31M), the RAM disc, and the font cache. More areas may become protected in future releases of RISC OS. The **only** areas you should directly access are the application workspace and the RMA. It is very dangerous to write to **any** other areas, or rely on certain locations containing given information, as these are subject to change. You should always use OS routines to access operating system workspace.

OS_ValidateAddress (SWI &3A) will check a range of logical addresses to see if they are mapped into physical memory.

Changing the logical map

The mapping that MEMC maintains from logical to physical address space can be read with OS_ReadMemMapEntries (SWI &52). This gives a list of physical addresses for a matching set of logical page numbers.

The reverse operation, OS_SetMemMapEntries (SWI &53) will write the mapping inside MEMC. Note that this is an extremely dangerous operation if you are not sure what you are doing.

OS_UpdateMEMC (SWI &1A) is a lower level operation that alters the bits in the MEMC control register.

Screen memory

Hardware scrolling is implemented by having the screen workspace at the end of logical memory, adjacent to the corresponding physical RAM banks which are mapped onto those addresses. This means that there are two adjacent copies of the screen memory as follows:

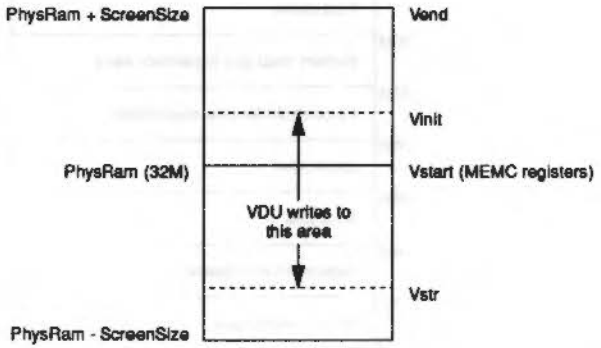


Figure 16.4 Screen memory

The screen can, therefore, be scrolled vertically by altering the VDU driver screen start address as shown above. This is usually performed automatically and you don't have to concern yourself with it.

OS_ClaimScreenMemory (SWI &41) allows you to claim or release this space.

The screen-size is configurable in units of one page (8K or 32K). Hence for a 20K screen on a 400 series machine, 32K will have to be used since it is the next highest multiple of 32K. For an 80K screen, 96K would be used, etc. In addition, if you want to use multiple banks of screen memory (eg for animation), enough memory must be reserved for each bank.

Because the total screen memory is often much more than is required at a given time, a facility is available whereby the 'extra' RAM can be claimed for short periods. It can be used as a buffer, in a data transfer operation, for example.

Non-volatile memory (CMOS RAM)

240 bytes of non-volatile memory are provided. Some of these are reserved since they hold default values for certain parameters and are set using various *Configure options. OS_Byte 161 allows you to read the CMOS memory directly, while OS_Byte 162 can write to it. The full list is given below:

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)
3	Econet printer server station id (0 ⇒ name configured)
4	Econet printer server net number (or first char of name)
5	Default filing system number
6 - 9	Reserved for Acorn use
10	Screen info:
	Bits 0 - 3 screen mode number. This is held in 5 bits The fifth bit is bit 1 in byte 133
	Bit 4 TV interlace (first *TV parameter)
	Bits 5 - 7 TV vertical adjust (signed three-bit number)
11	Shift, Caps mode:
	Bits 0 - 2 reserved
	Bits 3 - 5 ShCaps (001), NoCaps (010), Caps (100)
	Bit 6 - 7 reserved
12	Keyboard auto-repeat delay
13	Keyboard auto-repeat rate
14	Printer ignore character
15	Printer information:
	Bit 0 reserved
	Bit 1 0 ⇒ Ignore, 1 ⇒ NoIgnore
	Bits 2 - 4 serial baud rate (0=75,...,7=19200)
	Bits 5 - 7 printer type
16	Miscellaneous flags

	Bit 0 reserved
	Bit 1 0 ⇒ Quiet, 1 ⇒ Loud
	Bit 2 reserved
	Bit 3 0 ⇒ Scroll, 1 ⇒ NoScroll
	Bit 4 0 ⇒ NoBoot, 1 ⇒ Boot
	Bits 5 - 7 serial data format (0...7)
17 - 29	Reserved for Acorn use
30 - 45	Reserved for the user
46 - 79	Reserved for applications
80 - 111	Reserved for RISC iX
112 - 127	Reserved for expansion card use
128 - 129	Current year
130	Reserved for Acorn use
131	Reserved for Acorn use
132	DumpFormat
	Bits 0,1 control character print control
	00 print in GSTrans format
	01 print as a dot
	10 print decimal inside angle brackets
	11 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
	Bit 0 SyncBit
	Bit 1 top bit of mode configuration number in byte 10
	Bits 2 - 3 monitor type
134	FontSize in units of 4K
135 - 137	ADFS use
138 - 139	Set *Cat format
140 - 141	Set *Examine format
142	Twin's byte
143	Screen size in pagesize units.
144	RAM disc size in pagesize units
145	System heap size in pagesize to add after initialisation
146	RMA size in pagesize to add after initialisation
147	Sprite size in pagesize
148	SoundDefault parameters
	Bits 0 - 3 channel 0 default voice
	Bits 4 - 6 loudness (0 - 7 ⇒ &01, 13, 25, 37, 49, 5B, 6D,

7F)

Bit 7 loudspeaker enable

149 - 152 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 4 * 8 bits for unplugged modules in expansion cards
181 - 184 Wild card for BASIC editor
185 Configured language
186 Configured country
187 VFS
188 Bottom 2 bits are ROMFS Opt 4 state
189 - 192 Winchester size
193 Protection state

Bit 0 Peek
Bit 1 Poke
Bit 2 JSR
Bit 3 User RPC
Bit 4 OS RPC
Bit 5 Halt
Bit 6 GetRegs

194 Mouse multiplier
195 System speed – currently unused

Bits 0 - 3 RAM speed
Bit 4 ROM speed
Bit 5 Cache enable for ARM3
Bit 6 Broadcast protocols enable
Bit 7 Colour hourglass enable

196 Wimp mode (actual mode EOR &0C)
197 Wimp flags
198 Desktop state

Bits 0,1 display mode (Filer)
00 large icons
01 small icons
10 full info
11 reserved
Bits 2,3 sorting mode (Filer)
00 sort by name
01 sort by type
10 sort by size
11 sort by date
Bit 4 sorting mode (0 ⇒ name, 1 ⇒ number)
Bit 5 confirm option (1 ⇒ confirm)

Bit 6 verbose option (1 ⇒ verbose)
Bit 7 reserved

199 ADFS directory cache size
200 - 207 FontMax, FontMax1 - FontMax7
208 SCSIFS flags

Bits 0 - 2 number of discs (0 - 4)
Bits 3 - 5 default drive – 4
Bits 6,7 reserved

209 SCSIFS file cache buffers (must be 0)
210 SCSIFS directory cache size
211 - 214 SCSIFS disc sizes (their maps' sizes / 256)
224 - 238 Reserved for RISC iX
239 One byte for CMOS RAM checksum (not used in RISC OS 2)

Service Calls

Service_Memory (Service Call &11)

Memory controller about to be remapped

On entry

R0 = amount application space will change by
R1 = &11 (reason code)
R2 = current active object pointer (CAO)

On exit

R1 = 0 to prevent re-mapping taking place

Use

This is issued when the contents-addressable memory in the memory controller is about to be remapped, which alters the memory map of the machine. You should claim this call if you don't want the remapping to take place.

A module will initially be given the current slot size for its application workspace starting at &8000. However, modules do not generally need this area, as they use the RMA for workspace. Therefore, when a task calls Wimp_Initialise, the Wimp inspects the CAO. If this is within application workspace, the Wimp does nothing. However, if the CAO is outside of application space (a module's CAO is its base address in the RMA or ROM), the Wimp will reduce the current slot size to zero automatically, except as described below.

Some modules, notably BASIC, do require application workspace. Therefore the Wimp makes this service call just before returning the application space to its free pool. A task can object to the remapping taking place by claiming the call. The Wimp will then leave the application space as it is.

Service_MemoryMoved (Service Call &4E)

OS_ChangeDynamicArea has just finished.

On entry

R1 = &4E (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is made whenever OS_ChangeDynamicArea (SWI &2A) has just finished. It is used by the Wimp to tidy up and should never be claimed.

Service_ValidateAddress (Service Call &6D)

This service call is for internal use only. You must not use it in your own code.

SWI Calls

OS_Byte 161 (SWI &06)

Read battery-backed CMOS RAM

On entry

R0 = 161
R1 = RAM location

On exit

R0, R1 preserved
R2 = contents of location

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call provides read access to any of the locations in the battery-backed CMOS RAM. For example, this call may be used by a module to read a default configuration parameter. Moreover, this parameter could be examined by the user using the *Status command, if the module provides a suitable entry in its command decoding table. See the section entitled *Help and command keyword table* on page 1-207 for more details.

Related SWIs

OS_Byte 162 (SWI &06)

Related vectors

ByteV

OS_Byte 162 (SWI &06)

Write battery-backed CMOS RAM

On entry

R0 = 162
R1 = RAM location
R2 = value to be written

On exit

R0, R1 preserved
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call provides write access to any of the locations in the battery backed RAM with the exception of location zero, which is protected.

Related SWIs

OS_Byte 161 (SWI &06)

Related vectors

ByteV

OS_UpdateMEMC (SWI &1A)

Read or alter the contents of the MEMC control register

On entry

R0 = new bits in field
R1 = field mask

On exit

R0 = previous bits in field
R1 = previous field mask

Interrupts

Interrupts are disabled
Fast interrupts are disabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered because interrupts are disabled

Use

The memory controller (MEMC) chip is a write-only device. The operating system maintains a software copy of the current state of the control register and OS_UpdateMEMC updates MEMC from the software state. To allow the programming of individual bits the call takes a field and a mask. The new MEMC value is:

```
newMemC     = (oldMEMC AND NOT R1) OR (R0 AND R1)
R0          = oldMEMC
```

So to read the contents without altering them, R1 and R2 should both be zero. To set them to 'n', R1=&FFFFFFFF and R2=n.

Related SWIs

None

Related vectors

None

OS_Heap 0 (SWI &1D)

Initialise Heap

On entry

R0 = 0 (reason code)
R1 = pointer to heap to initialise
R3 = size of heap

On exit

R0, R1, R3 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call checks the given heap pointer, and then writes a valid descriptor into the heap it points at. The heap is then ready for use. The value given for R1 must be word-aligned and less than 32Mbytes (ie must point to an area of logical RAM). R3 must be a multiple of four and less than 16Mbytes.

Related SWIs

None

Related vectors

None

OS_Heap 1 (SWI &1D)

Describe Heap

On entry

R0 = 1 (reason code)
R1 = pointer to heap

On exit

R0, R1 preserved
R2 = largest available block size
R3 = total free

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call returns information on the space available in the heap. An error is returned if the heap is invalid. This may be for any of the following reasons:

- the heap descriptor is corrupt
- the information within the heap is not sensible
- R1 does not point to a heap

Related SWIs

None

Related vectors

None

OS_Heap 2 (SWI &1D)

Get heap block

On entry

R0 = 2 (reason code)
R1 = pointer to heap
R3 = size required in bytes

On exit

R0, R1 preserved
R2 = pointer to claimed block or zero if allocation failed
R3 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This allocates a block from the heap. An error is returned if the allocation failed for any of the following reasons:

- there is not a large enough block left in the heap
- the heap has been corrupted
- R1 does not point to a heap

Related SWIs

None

Related vectors

None

OS_Heap 3 (SWI &1D)

Free heap block

On entry

R0 = 3 (reason code)
R1 = pointer to heap
R2 = pointer to block

On exit

R0 - R2 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This checks that the pointer given refers to an allocated block in the heap, and deallocates it. Deallocation tries to join free blocks together if at all possible, but if the block being freed is not adjacent to any other free block it is just added to the list of free blocks. An error is returned if the deallocation failed which may be because:

- R1 does not point to a heap
- the heap descriptor or heap was corrupted
- R2 does not point to an allocated block in the heap.

Related SWIs

None

Related vectors

None

OS_Heap 4 (SWI &1D)

Extend heap block

On entry

R0 = 4 (reason code)
R1 = pointer to heap
R2 = pointer to block
R3 = required size change in bytes (signed integer)

On exit

R0, R1 preserved
R2 = new block pointer, or -1 if heap block extended to size 0 (or less)
R3 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This attempts to enlarge or shrink the given block in its current position if possible, or, if this is not possible, by reallocating and copying it. Note that if the block has to be moved, it is your responsibility to note this (by the fact that R2 has been altered), and to perform any necessary relocation of data within the block.

Related SWIs

None

Related vectors

None

OS_Heap 5 (SWI &1D)

Extend heap

On entry

R0 = 5 (reason code)
R1 = pointer to heap
R3 = required size change in bytes (signed integer)

On exit

R0, R1, R3 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This updates the heap size information to take account of the new size. An error is returned if it cannot shrink far enough, because of data that has already been allocated.

Related SWIs

None

Related vectors

None

OS_Heap 6 (SWI &1D)

Read block size

On entry

R0 = 6 (reason code)
R1 = pointer to heap
R2 = pointer to block

On exit

R0 - R2 preserved
R3 = current block size

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This reads the size of a block in the specified heap. An error is returned if the heap or the block could not be found.

Related SWIs

None

Related vectors

None

OS_ChangeDynamicArea (SWI &2A)

Alter the space allocation of a dynamic area

On entry

R0 = area to alter
R1 = amount to move in bytes (signed integer)

On exit

R0 = preserved
R1 = number of bytes moved (unsigned integer)

Interrupts

Interrupts are enabled, or interrupt status is not altered (RISC OS 2.0)
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_ChangeDynamicArea allows the space allocated to an area to be altered in size by removing or adding workspace from the application workspace.

The area to be altered depends on R0 as follows:

Value of R0	Area to alter
0	system heap
1	RMA
2	screen area
3	sprite area
4	font cache
5	RAM filing system

The amount to move is given by the sign and magnitude of R1:

+ve means **enlarge** the selected area by at least the given amount
–ve means **shrink** the selected area by no more than the given amount

If the amount to be moved is not an exact number of pages, it is rounded up (ie in the +ve direction) to the next number of pages.

Note that normally, this cannot be used while the application work area is being used; for example when a language is active outside the RISC OS desktop. An attempt to do so will result in a 'Memory in use' error. (In fact, when this call is made, the OS passes a service call round to modules, which can veto the change if they can't handle it correctly. See *Service_Memory* (*Service Call* &11) on page 1-350 and *Service_MemoryMoved* (*Service Call* &4E) on page 1-351 for more details.

Any area size change will fail if the new size is smaller than the current requirements, but will shrink the area as far as it can. If you need to release as much space as possible from an area, try to reduce its size by 16 Mbytes.

Expanding, on the other hand, does nothing if it can't move enough. In this case, if you asked for the extra space you probably need it all; RISC OS assumes that half the job is no use to you.

This SWI also does an UpCall, to enable programs running in application workspace to allow movement of memory. If the UpCall is claimed when the application is running in application workspace, the memory movement is allowed to proceed. For full details see OS_*UpCall* 257 (SWI &33) on page 1-187.

An error is returned if not all the bytes were moved, or if application workspace is being used – ie an application is active.

Related SWIs

OS_ReadDynamicArea (SWI &5C)

Related vectors

None

OS_ValidateAddress (SWI &3A)

Check that a range of addresses are in logical RAM

On entry

R0 = minimum address
R1 = maximum address

On exit

R0, R1 preserved
C flag is clear if the range is OK, set otherwise

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI checks the address range between R0 and R1 minus 1 to see if they are valid. If they are equal, then that single address is checked. Valid addresses are in logical RAM (0 - 32M) and have a mapping into physical RAM, including screen RAM, throughout the specified range.

Related SWIs

None

Related vectors

None

OS_ClaimScreenMemory (SWI &41)

Use spare screen memory

On entry

R0 = 0 for release, 1 for claim
R1 = length required in bytes (if R0 = 1)

On exit

R0 preserved
if the C flag is 0, then memory was claimed successfully
    R1 = length available
    R2 = start address
if the C flag is 1, then memory could not be claimed
    R1 = length that is available

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

There are several restrictions to the use of screen memory. It can only be claimed by one 'client' at a time, who gets all of it. It can only be claimed if no bank other than bank 1 has been used. You can't claim it, for example, if the shadow bank has been used.

While you have claimed the screen memory, you must not perform any action which might causes the screen to scroll. This means avoiding the use of routines which might cause screen output.

It is important to release the memory after it has been used.

Related SWIs

None

Related vectors

None

OS_ReadRAMFsLimits (SWI &4A)

Get the current limits of the RAM filing system

On entry

—

On exit

R0 = start address
R1 = end address + 1 byte

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This reads the start and end addresses of the RAM filing system. This information can also be read from OS_ReadDynamicArea.

If the RamFS is configured to zero size then R0 and R1 have the same value on exit.

The size of the RamFS after a hard reset (ie the difference between the two return values) can be configured using *Configure RamFsSize.

Related SWIs

OS_ReadDynamicArea (SWI &5C)

Related vectors

None

OS_DelinkApplication (SWI &4D)

Remove any vectors that an application is using

On entry

R0 = pointer to buffer
R1 = buffer size in bytes

On exit

R0 preserved
R1 = number of bytes left in buffer

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entrant because interrupts are disabled

Use

When an application running at &8000 is going to be swapped out, it must remove all vectors that it uses. Otherwise, if they were activated, they would jump into whatever happened to be at that location in the new application running in that space.

R0 on entry points to a buffer. This is used to store details of the vectors used, so that they can be restored afterwards. Each vector requires 12 bytes of storage and the list is terminated by a single byte.

If the space left returned in R1 is zero, then you must allocate another buffer and repeat the call; the buffer you have contains valid information. When you relink you must pass all the buffers returned by this call.

Related SWIs

OS_RelinkApplication (SWI &4E)

Related vectors

None

OS_RelinkApplication (SWI &4E)

Restore any vectors that an application is using from a buffer

On entry

R0 = pointer to buffer

On exit

R0 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

When an application is going to be swapped in, all vectors that it uses must be restored.

R0 on entry points to a buffer, which has previously been created by OS_DelinkApplication.

Related SWIs

OS_DelinkApplication (SWI &4D)

Related vectors

None

OS_ReadMemMapInfo (SWI &51)

Read the page size and count

On entry

—

On exit

R0 = page size in bytes
R1 = number of pages

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the page size used by MEMC and the number of pages in use. The valid page numbers are 0 to R1 - 1, and the total memory size is R0 times R1 bytes.

Related SWIs

None

Related vectors

None

OS_ReadMemMapEntries (SWI &52)

Read the logical to physical memory mapping used by MEMC

On entry

R0 = pointer to request list

On exit

R0 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the logical to physical memory mapping used by MEMC. For given page numbers, it finds the corresponding logical address and protection level.

The request list is a series of entries three words long, terminated by a –1 in the first word. The three words are used for:

Word	Meaning
1	page number (from 0 upwards)
2	logical address that it is mapped to
3	protection level

This is a bitfield, which uses the bottom 2 bits:

00	readable and writable by everybody
01	read-only in user mode
10	inaccessible in user mode.

All other bits are reserved and must be written as zero.

On entry, the page number fields must be set; on exit, all fields are set.

Related SWIs

OS_SetMemMapEntries (SWI &53), OS_FindMemMapEntries (SWI &60)

Related vectors

None

OS_SetMemMapEntries (SWI &53)

Write the logical to physical memory mapping used by MEMC

On entry

R0 = pointer to request list

On exit

R0 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call writes the logical to physical memory mapping used by MEMC.

The request list is a series of entries three words long, terminated by a -1 in the first word. The three words are used for:

Word	Meaning
1	page number (from 0 upwards)
2	logical address that it is mapped to
3	protection level

This is a bitfield, which uses the bottom 2 bits:

00	readable and writable by everybody
01	read-only in user mode
10	inaccessible in user mode.

All other bits are reserved and must be written as zero.

All fields must be set on entry.

Any address above 32Mbyte (&2000000) makes that page inaccessible. This also sets the protection level to minimum accessibility.

This SWI assumes you know what you are doing. It will set any page to any address, with no checks at all.

If you are using this call, then you can only use OS_ChangeDynamicArea if the kernel's limits are maintained, and all appropriate areas contain continuous memory.

Related SWIs

OS_ChangeDynamicArea (SWI &2A), OS_ReadMemMapEntries (SWI &52), OS_FindMemMapEntries (SWI &60)

Related vectors

None

OS_ReadDynamicArea (SWI &5C)

Read the space allocation of a dynamic area

On entry

R0 = area to read

On exit

R0 = pointer to start of area
R1 = current number of bytes in area

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI reads the size of an area. The area read depends on R0 as follows:

Value of R0	Area to read
0	system heap
1	RMA
2	screen area
3	sprite area
4	font cache
5	RAM filing system

Related SWIs

OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

OS_FindMemMapEntries (SWI &60)

Read the logical to physical memory mapping used by MEMC

On entry

R0 = pointer to request list

On exit

R0 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the logical to physical memory mapping used by MEMC. For given logical address, it finds the corresponding page number and protection level.

The request list is a series of entries three words long, terminated by a -1 in the first word. The three words are used for:

Word	Meaning
1	page number (from 0 upwards)
2	logical address that it is mapped to
3	protection level

This is a bitfield, which uses the bottom 2 bits:

00	readable and writable by everybody
01	read-only in user mode
10	inaccessible in user mode.

All other bits are reserved and must be written as zero.

On entry, the logical address fields must be set. You may supply probable page numbers, which (if correct) will make this call return more quickly than it might otherwise. If you have no idea what the page number might be, you should set the page number to zero on entry. The protection value is ignored on entry.

If the page number is –1 on exit, then the memory map entry was not found; in this case, the protection level will always be 3. Otherwise the request list has been updated with the page number and protection level for the given logical address.

This call is not available in RISC OS 2.0.

Related SWIs

OS_ReadMemMapEntries (SWI &52), OS_SetMemMapEntries (SWI &53)

Related vectors

None

*Commands

*Configure

Sets the value of a configuration option in the CMOS RAM

Syntax

```
*Configure [option [value]]
```

Parameters

option	the name of a configuration option
value	its new value(s)

Use

*Configure sets the value of a configuration option in the CMOS RAM. These are used to permanently store the *configuration* (or set-up) of the computer. They are made current on initial power-on and after a hard break (Ctrl-Reset), and do **not** take effect immediately.

If no parameters are specified, the available configuration options are listed.

If parameters are specified, the given value is stored in the location in CMOS RAM appropriate for the given option. Some options require more than one value, and some require none at all.

Where a number is required, you may give it in decimal, as a hex number preceded by &, or a number of the form *base_num*, where *base* is the base of the number in decimal in the range 2 to 36. For example 2_1010 is another way of saying 10.

Here is a list of the available configuration options, the details of which can be found on the appropriate pages:

User Preferences	In the chapter	on page
*Configure Boot	*FileSwitch*	page 3-141
*Configure Caps	*Character Input*	page 2-422
*Configure Delay	*Character Input*	page 2-423
*Configure Dir	*FileCore*	page 3-239
*Configure DumpFormat	*FileSwitch*	page 3-142
*Configure FileSystem	*FileSwitch*	page 3-144
*Configure FontMax1	*The Font Manager*	page 5-91
*Configure FontMax2	*The Font Manager*	page 5-93
*Configure FontMax3	*The Font Manager*	page 5-95
*Configure FontMax4	*The Font Manager*	page 5-97

*Configure FontMax5	*The Font Manager*	page 5-97
*Configure Language	*The rest of the kernel*	page 2-454
*Configure Lib	NetFS	page 3-356
*Configure Loud	VDU Drivers	page 2-230
*Configure Mode	VDU Drivers	page 2-231
*Configure MouseStep	VDU Drivers	page 2-234
*Configure NoBoot	*FileSwitch*	page 3-145
*Configure NoCaps	*Character Input*	page 2-424
*Configure NoDir	*FileCore*	page 3-240
*Configure NoScroll	VDU Drivers	page 2-235
*Configure Quiet	VDU Drivers	page 2-236
*Configure Repeat	*Character Input*	page 2-425
*Configure Scroll	VDU Drivers	page 2-238
*Configure ShCaps	*Character Input*	page 2-426
*Configure SoundDefault	*The Sound system*	page 5-384
*Configure Truncate	*FileSwitch*	page 3-146
*Configure WimpAutoMenuDelay	*The Window Manager*	page 4-320
*Configure WimpDoubleClickDelay	*The Window Manager*	page 4-321
*Configure WimpDoubleClickMove	*The Window Manager*	page 4-322
*Configure WimpDragDelay	*The Window Manager*	page 4-323
*Configure WimpDragMove	*The Window Manager*	page 4-324
*Configure WimpFlags	*The Window Manager*	page 4-325
*Configure WimpMenuDragDelay	*The Window Manager*	page 4-327
*Configure WimpMode	*The Window Manager*	page 4-328

Hardware configuration	**In the chapter**	**on page**
*Configure Baud	*Serial device*	page 3-454
*Configure Country	*International module*	page 5-272
*Configure Data	*Serial device*	page 3-456
*Configure Drive	ADFS	page 3-288
*Configure DST	*The Territory Manager*	page 5-330
*Configure Floppies	ADFS	page 3-289
*Configure FS	NetFS	page 3-355
*Configure HardDiscs	ADFS	page 3-290
*Configure IDEDiscs	ADFS	page 3-290
*Configure Ignore	*Character Output*	page 2-36
*Configure MonitorType	VDU Drivers	page 2-232
*Configure NoDST	*The Territory Manager*	page 5-331
*Configure Print	*Character Output*	page 2-37
*Configure PS	NetPrint	page 3-380
*Configure Step	ADFS	page 3-292
*Configure Sync	VDU Drivers	page 2-239
*Configure Territory	*The Territory Manager*	page 5-332
*Configure TV	VDU Drivers	page 2-240

Memory allocation	**In the chapter**	**on page**
*Configure ADFSbuffers	ADFS	page 3-286
*Configure ADFSDirCache	ADFS	page 3-287
*Configure FontMax	*The Font Manager*	page 5-89
*Configure FontSize	*The Font Manager*	page 5-101
*Configure RamFsSize	*RamFS*	page 3-303
*Configure RMASize	*Memory Management*	page 1-388
*Configure ScreenSize	VDU Drivers	page 2-237
*Configure SpriteSize	*Sprites*	page 2-320
*Configure SystemSize	*Memory Management*	page 1-389

Example

```
*Configure Baud 7
```

Related commands

*Status

Related SWIs

None

Related vectors

None

*Configure RMASize

Sets the configured extra area of memory reserved for relocatable modules

Syntax

```
*Configure RMASize mK|n
```

Parameters

mK	number of kilobytes of memory reserved
n	number of pages of memory reserved; n ≤ 127

Use

*Configure RMASize sets the configured extra area of memory reserved in the relocatable module area (RMA) after all modules have been initialised. The default is to reserve 2 extra pages.

If the parameter is 0, no extra memory is reserved.

Example

```
*Configure RMASize 128K
```

Related commands

None

Related SWIs

OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

*Configure SystemSize

Sets the configured extra area of memory reserved for the system heap

Syntax

```
*Configure SystemSize mK|n
```

Parameters

mK	number of kilobytes of memory reserved
n	number of pages of memory reserved; n ≤ 63

Use

*Configure SystemSize sets the configured extra area of memory reserved for the system heap after all modules have been initialised. The default value is 0.

Example

```
*Configure SystemSize 32K
```

Related commands

None

Related SWIs

OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

*Status

Provides information on how the computer is configured

Syntax

```
*Status [option]
```

Parameters

option	the name of a configuration option

Use

*Status displays the value of a configuration option in the CMOS RAM. If no option is specified, the values of all configuration options are shown.

Because the values of these configuration options are held in non-volatile memory (the battery-backed CMOS RAM) they are preserved even when the computer is switched off, until reset by using either the Configure application from the desktop or the *Configure command from the command line.

Example

*Status TV

Related commands

*Configure

Related SWIs

None

Related vectors

None

17 Time and Date

Introduction

There are two basic aspects of time dealt with in this chapter: passive aspects such as reading various clock settings; and active ones, where an event occurs when a given time is reached. In this chapter, a *clock* is a place where a stored value is incremented on a regular basis. The *time* is the name of the value as it is read or written.

There are several clocks that increment every 1/100th of a second (centisecond). One of them cannot be changed except by a hard reset. This is useful for time-stamping events, such as mouse moves. Another can be changed by a program, so is useful for elapsed time calculations.

The real-time clock keeps the real-world time, and represents time in centiseconds since 00:00:00 on January 1 1900. There are calls to present this information in a number of ways. The real-time can be converted to a string with complete program control over its format.

A variety of timer events can be set up. There are SWIs that will call your application after a given delay has passed or every time that delay has elapsed. You can set up a routine to sit on the ticker vector, to enable it to be called every centisecond.

A specialised form of timer event is one that will occur every time the screen driving hardware reaches the bottom of the screen. This event is useful for flicker-free redrawing. See the chapter entitled VDU Drivers on page 2-39 for further details.

Overview and Technical Details

There are four timers, which increment at a centisecond rate. They are:

- the monotonic timer (read-only)
- the system timer (read/write)
- the interval time (read/write)
- the real-time clock (read only in general – ie only users should change it).

Monotonic timer

A monotonic timer cannot be written, except by a hard reset or when the machine is turned on. OS_ReadMonotonicTime (SWI &42) allows you to read this value. It is useful for time-stamping within an application, such as event times. Because it can never be changed, the order of events cannot be confused.

It is stored as a 4-byte value with least significant byte first. It is incremented every centisecond, which means that it would take nearly 500 days for it to wrap around.

System clock

The system clock is stored as a 5-byte value. Like the monotonic timer it is reset by hard resets and increments every centisecond. However it can be altered. This is useful for measuring elapsed times in an application. OS_Word 1 reads the value and OS_Word 2 writes it.

Real-time

The real-time clock is stored as a 5-byte value in the CMOS clock chip and reflects the normal usage of the word clock. That is, it stores the elapsed number of centiseconds since 00:00:00 on January 1 1900. You can set it using the Clock or Alarm applications on the desktop.

Under RISC OS 2.0 the real-time clock is assumed to be set to local time. Under later versions, the real-time clock is assumed to be set to UTC, or *Universal Time Coordinated*. (This is the same as GMT, or Greenwich Mean Time.) Territory modules provide the necessary information for the kernel to convert the real-time clock value to a local time in a suitable format.

A soft-copy of the real time clock is also kept by RISC OS and is used by the filing system to date-stamp files. This soft-copy is updated from the CMOS clock chip following a hard reset

String format

*Time displays the local time and date as a string. It calls OS_Word 14,0 to do so. The format of the string depends on the territory which the computer is set to use. (It is fixed in RISC OS 2.0, which does not support territories.) For example:

```
Tue,28 Mar 1989.13:25:54
```

5-byte format

The real-time clock can be read in the standard 5-byte format using OS_Word 14,3. This, or any, 5-byte time can be converted into a string using OS_ConvertStandardDateAndTime (SWI &C0).

Changing real-time

The real-time clock's time of day can be altered with OS_Word 15,8, its date with OS_Word 15,15, or both with OS_Word 15,24. These calls all use the time in a string format (see above).

Format field names

The above time string is not flexible. You can customise the way that the time and date is presented by supplying a format string. The string is copied character for character to the output buffer unless a '%' is found. If this character is followed by any of the following codes, then the appropriate value is copied to the output buffer.

Name	Value	Examples
CS	Centi-seconds	99
SE	Seconds	59
MI	Minutes	05
12	Hours in 12 hour format	07
24	Hours in 24 hour format	23
AM	AM or PM indicator (in local language)	PM
PM	AM or PM indicator (in local language)	AM
WE	Weekday – full (in local language)	Thursday
W3	Weekday – short (in local language)	Thu
WN	Weekday – number	5
DY	Day of the month (in local language)	01
ST	Ordinal pre/suffix (in local language)	st nd rd th
MO	Month name – full (in local language)	September
M3	Month name – short (in local language)	Sep
MN	Month – number	09
CE	Century	19
YR	Year within century	87
WK	Week of year (using local start of week)	52
DN	Day of the year	364
0	Insert an ASCII 0 zero byte	
%	Insert a '%'	

You must not make **any** assumptions about the nature or length of any fields that use the local language. For example: short forms of the weekday or month are three characters long in the UK territory, but may have a different length in other territories; the day of the month may not be numeric; ordinals may be null; and so on...

To cause leading zeros to be omitted, prefix the field with the letter Z. For example, %zmn means the month number without leading zeros. %0 may be used to split the output into several zero-terminated strings.

As an example, this format string:

```
%W3,%DY %M3 %CE%YR.%24:%MI:%SE
```

would produce this time string in the UK territory:

```
Tue,28 Mar 1989.13:25:54
```

OS_ConvertDateAndTime (SWI &C1) will convert a 5-byte time into a string using a supplied format string.

BCD conversions

The CMOS clock chip stores the time internally in a Binary Coded Decimal (BCD) format. OS_Word 14,1 will read the time as a 7-byte BCD block. OS_Word 14,2 will convert this BCD block into a string.

Timer events

There are three different causes of timer events: the interval timer, the timer chain and the VSync timer.

Interval timer

The interval timer is a 5-byte clock that increments every centisecond. If enabled by OS_Byte 14, an event will occur when the counter reaches zero. Thus to wait for a given time, the interval timer must be set to the negative of it using OS_Word 4. OS_Word 3 can read the current setting of the interval timer.

For example, to wait 10 seconds, -1000 must be passed to OS_Word 4.

The interval timer is kept for compatibility with earlier Acorn operating systems. Its use should be avoided if possible. It is especially important that this is not used under the Wimp, since it cannot cope with more than one program using it at once.

Timer chain

An easier to use and more sophisticated way for an application to be called at a given time is the timer chain. These are independent of event routines, but are used in a similar manner. OS_CallAfter (SWI &3B) can be used to get a given address to be called after a certain time has elapsed. OS_CallEvery (SWI &3C) is like this, but automatically reloads the counter when it has expired. OS_RemoveTickerEvent (SWI &3D) will cancel either OS_CallAfter before it occurs or OS_CallEvery to stop it repeating forever.

OS_CallAfter and OS_CallEvery are passed an address to call, the delay to wait and an identification word to return in R12. Thus, many timers can be running concurrently.

These are stored in a list which can be any size up to the machine memory limit.

VSync timer

The screen is refreshed 50 times a second in Standard monitor type modes. From the time that the bottom of the screen is complete till the top of the screen commences again is a delay called the vertical sync period. This allows the electron

beam to go to this start position. The VSync event coincides with the vertical sync beginning. You can use OS_Byte 14 to enable this event, so that flicker-free re-drawing can be done while the VDU is not being written to.

OS_Byte 176 provides access to a one byte counter in 50Hz periods; ie it decrements at the rate of the VSync event.

Obsolete timers

OS_Byte 243 reads a temporary location used by the timer software. It is kept for compatibility with earlier Acorn operating systems and must not be used.

SWI Calls

OS_Byte 176 (SWI &06)

Read/Write 50Hz counter

On entry

R0 = 176
R1 = 0 to read or new value to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call reads or writes a one-byte counter which is decremented at a 50Hz rate; or more precisely at the rate of the VSync interrupt.

Related SWIs

None

Related vectors

ByteV

OS_Byte 243 (SWI &06)

Read timer switch state

On entry

R0 = 243
R1 = 0
R2 = 255

On exit

R0 preserved
R1 = switch state
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

In order to protect the centi-second clock against corruption during reset, the OS keeps two copies. One of them is the one which will be read or written when one of the OS_Words is called, the other is the one which will be updated during the next 100Hz interrupt. When the update has been performed correctly, the values are swapped. This OS_Byte enables you to read the byte which indicates which copy is being used. Its only practical use is as a location which changes 100 times a second.

This call is obsolete and should not be used.

Related SWIs

OS_Word 3 (SWI &07), OS_Word 4 (SWI &07)

Related vectors

ByteV

OS_Word 1 (SWI &07)

Read system clock

On entry

R0 = 1
R1 = pointer to five byte block

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains the value of the system clock at the instant of the call.

R1+0 = time (least significant byte)
R1+1 = ...
R1+2 = ...
R1+3 = ...
R1+4 = time (most significant byte)

The clock is incremented every centi-second. The value of the clock is preserved over a soft break and set to zero after a hard break.

Related SWIs

OS_Word 2 (SWI &07)

Related vectors

WordV

OS_Word 2 (SWI &07)

Write system clock

On entry

R0 = 2
R1 = pointer to five byte block with centi-second clock value in it

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On entry, the parameter block contains the value to set the system clock.

R1+0 = time (least significant byte)
R1+1 = ...
R1+2 = ...
R1+3 = ...
R1+4 = time (most significant byte)

This allows the clock to be set to a specified value.

Related SWIs

OS_Word 1 (SWI &07)

Related vectors

WordV

OS_Word 3 (SWI &07)

Read interval timer

On entry

R0 = 3
R1 = pointer to five byte block

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains the value of the interval timer at the instant of the call.

R1+0 = time (least significant byte)
R1+1 = ...
R1+2 = ...
R1+3 = ...
R1+4 = time (most significant byte)

Like the system clock, the interval timer is incremented 100 times a second. The interval timer can be made to cause an event when its value reaches zero. To do this, it must be set to minus the number of centi-seconds that are to elapse before the event takes place.

To produce repeated events, the routine servicing the timer event should reload the timer with the appropriate number. For example, to produce an event every 10 seconds, reload it with -1000 (&FFFFFC18). An alternative is to use the special ticker event, described in the chapter entitled Events on page 1-137.

Related SWIs

OS_Word 4 (SWI &07)

Related vectors

WordV

OS_Word 4 (SWI &07)

Write interval timer

On entry

R0 = 4
R1 = pointer to five byte block

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On entry, the parameter block contains the value to set the interval timer.

R1+0 = time (least significant byte)
R1+1 = ...
R1+2 = ...
R1+3 = ...
R1+4 = time (most significant byte)

This call resets the interval timer to a specified value.

Note that you must use OS_Byte 14 to enable the interval timer event.

Related SWIs

OS_Word 3 (SWI &07)

Related vectors

WordV

OS_Word 14,0 (SWI &07)

Read soft copy of the real-time clock as a string, converting to local time

On entry

R0 = 14
R1 = pointer to parameter block
R1+0 = 0 (reason code)

On exit

R0, R1 preserved

Interrupts

Interrupts are enabled (in RISC OS 2.0, the interrupt status is not altered)
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains the local time as a string terminated by a Return character (ASCII 13). The format of the string depends on the territory which the computer is set to use. (It is fixed in RISC OS 2.0, which does not support territories.)

This time string comes from the soft copy of the real-time clock maintained by RISC OS, rather than from the CMOS clock chip itself.

This call is equivalent to the *Time command.

Related SWIs

OS_Word 15 (SWI &07)

Related vectors

WordV

OS_Word 14,1 (SWI &07)

Read time from CMOS clock chip in Binary Coded Decimal (BCD) format, converting to local time

On entry

R0 = 14
R1 = pointer to parameter block
R1+0 = 1 (reason code)

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains a seven-byte BCD clock value:

R1+0 = year	(00 - 99)
R1+1 = month	(01 - 12; 01 = January etc)
R1+2 = day of month	(01 - 31)
R1+3 = day of week	(01 - 07; 01 = Sunday etc)
R1+4 = hours	(00 - 23)
R1+5 = minutes	(00 - 59)
R1+6 = seconds	(00 - 59)

The clock value is read directly from the CMOS clock chip. Under RISC OS 2.0 the real-time clock is assumed to be set to local time, and so the value is not further converted. Under later versions of RISC OS the real-time clock is assumed to be set to UTC, and so the value is then converted to local time.

Related SWIs

OS_Word 15 (SWI &07)

Related vectors

WordV

OS_Word 14,2 (SWI &07)

Convert BCD clock value into string format

On entry

R0 = 14
R1 = pointer to parameter block

R1+0 = 2	reason code
R1+1 = year	(00 - 99)
R1+2 = month	(01 - 12; 01 = January etc)
R1+3 = day of month	(01 - 31)
R1+4 = day of week	(01 - 07; 01 = Sunday etc)
R1+5 = hours	(00 - 23)
R1+6 = minutes	(00 - 59)
R1+7 = seconds	(00 - 59)

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On entry, the parameter block contains a 7-byte BCD clock value:

On exit, the parameter block contains a string terminated by a Return character (ASCII 13), representing the same time. The format of the string depends on the territory which the computer is set to use. (It is fixed in RISC OS 2.0, which does not support territories.)

Related SWIs

OS_Word 15 (SWI &07)

Related vectors

WordV

OS_Word 14,3 (SWI &07)

Read real-time in 5-byte format

On entry

R0 = 14
R1 = pointer to parameter block
    R1+0 = 3 (reason code)

On exit

R0 preserved
R1 preserved:
    R1+0 = LSB of time
    R1+1 = ...
    R1+2 = ...
    R1+3 = ...
    R1+4 = MSB of time

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit the parameter block contains the 5-byte real time read directly from the soft copy of the real-time clock. This number gives the elapsed number of centiseconds since 00:00:00 on January 1 1900. Under RISC OS 2.0 the real-time clock is assumed to be set to local time; under later versions the real-time clock is assumed to be set to UTC.

This 5-byte real-time is used for time/date stamping by the filing system. It is also useful for utilities which are used for building consistent systems, eg 'Make'.

Related SWIs

OS_Word 15 (SWI &07)

Related vectors

WordV

OS_Word 15,8 (SWI &07)

Writes the time of day to both the CMOS clock and its soft copy

On entry

R0 = 15
R1 = pointer to parameter block
R1+0 = 8 (reason code)
R1+1... = string giving time of day (in local language)

On exit

R0, R1 preserved

The C flag will be set on exit, if the parameter block contained a format error.

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call writes the time of day to both the CMOS clock and its soft copy.

On entry, the parameter block contains the local time of day as a string; the format this string must have depends on the territory which the computer is set to use. (It is fixed in RISC OS 2.0, which does not support territories.)

Related SWIs

OS_Word 14 (SWI &07)

Related vectors

WordV

OS_Word 15,15 (SWI &07)

Writes the date to both the CMOS clock and its soft copy

On entry

R0 = 15
R1 = pointer to parameter block
R1+0 = 15 (reason code)
R1+1... = string giving date (in local language)

On exit

R0, R1 preserved

The C flag will be set on exit, if the parameter block contained a format error.

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call writes the date to both the CMOS clock and its soft copy.

On entry, the parameter block contains the local date as a string; the format this string must have depends on the territory which the computer is set to use. (It is fixed in RISC OS 2.0, which does not support territories.)

Related SWIs

OS_Word 14 (SWI &07)

Related vectors

WordV

OS_Word 15,24 (SWI &07)

Writes the time of day and date to both the CMOS clock and its soft copy

On entry

R0 = 15
R1 = pointer to parameter block
R1+0 = 24 (reason code)
R1+1... = string giving time of day and date (in local language)

On exit

R0, R1 preserved

The C flag will be set on exit, if the parameter block contained a format error.

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call writes the time of day and date to both the CMOS clock and its soft copy.

On entry, the parameter block contains the local time of day and date as a string; the format this string must have depends on the territory which the computer is set to use. (It is fixed in RISC OS 2.0, which does not support territories.)

Related SWIs

OS_Word 14 (SWI &07)

Related vectors

WordV

OS_CallAfter (SWI &3B)

Call a specified address after a delay

On entry

R0 = time in centi-seconds
R1 = address to call
R2 = value of R12 to call code with

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_CallAfter calls the code pointed to by R1 after the delay specified in R0. The code should regard itself as an interrupt routine, and behave accordingly.

OS_RemoveTickerEvent can be used to cancel a pending OS_CallAfter

In RISC OS 2.0 this call may return incorrect error pointers. An invalid value of R0 now generates the error message 'Invalid time interval', rather than the null string generated by RISC OS 2.0.

Related SWIs

OS_CallEvery (SWI &3C), OS_RemoveTickerEvent (SWI &3D)

Related vectors

None

OS_CallEvery (SWI &3C)

Call a specified address every time a delay elapses

On entry

R0 = (delay in centi-seconds) - 1
R1 = address to call
R2 = value of R12 to call code with

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_CallEvery calls the code pointed to by R1 every (R0+1)centiseconds, until OS_RemoveTickerEvent is executed or Break is pressed. The code should regard itself as an interrupt routine, and behave accordingly. The minimum value for R0 is 1, which means the minimum possible delay is 2 centiseconds. If you wish to be called every centisecond, you must instead claim TickerV.

In RISC OS 2 this call may return incorrect error pointers. An invalid value of R0 now generates the error message 'Invalid time interval', rather than the null string generated by RISC OS 2.

Related SWIs

OS_CallAfter (SWI &3B), OS_RemoveTickerEvent (SWI &3D)

Related vectors

None

OS_RemoveTickerEvent (SWI &3D)

Remove a given call address and R12 value from the ticker event list

On entry

R0 = call address
R1 = value of R12 used in OS_CallEvery or OS_CallAfter

On exit

R0, R1 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_RemoveTickerEvent takes R0 as the address and R1 as the R12 value of the event to find and remove from its list.

It is used to stop an event set up by a call to OS_CallAfter or OS_CallEvery. The parameters passed must match those originally passed to OS_CallEvery or OS_CallAfter for it to remove the correct event.

Related SWIs

OS_CallAfter (SWI &3B), OS_CallEvery (SWI &3C)

Related vectors

None

OS_ReadMonotonicTime (SWI &42)

Number of centi-seconds since the last hard reset

On entry

—

On exit

R0 = time in centi-seconds

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadMonotonicTime returns the number of centi-seconds since the last hard reset, or switching on of the machine. 'Monotonic' refers to the fact that this timer is guaranteed to increase with time. It is used, for example, to time-stamp mouse events.

Related SWIs

None

Related vectors

None

OS_ConvertStandardDateAndTime (SWI &C0)

Converts a 5-byte time into a string

On entry

R0 = pointer to 5-byte time block
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating zero in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ConvertStandardDateAndTime converts a five-byte value representing the number of centi-seconds since 00:00:00 on January 1st 1900 into a string. It converts it using a standard format string stored in the system variable 'SysSDateFormat' and places it in a buffer (which should be at least 20 bytes).

For details of the format field names see the section entitled *Format field names* on page 1-393.

Related SWIs

OS_ConvertDateAndTime (SWI &C1)

Related vectors

None

OS_ConvertDateAndTime (SWI &C1)

Convert 5-byte time into a string using a supplied format string

On entry

R0 = pointer to 5-byte time block
R1 = pointer to buffer for resulting string
R2 = size of buffer
R3 = pointer to format string (null terminated)

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating zero in buffer
R2 = number of free bytes in buffer
R3 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ConvertDateAndTime converts a five byte value representing the number of centi-seconds since 00:00:00 on January 1st 1900 into a string. It converts it using the format string supplied.

Apart from the following exception, the format string is copied directly into the result buffer. However, whenever '%' appears in the format string, the next two characters are treated as a special field name which is replaced by a component of the current time.

For details of the format field names see the section entitled *Format field names* on page 1-393.

Related SWIs

OS_ConvertStandardDateAndTime (SWI &C0)

Related vectors

None

*Commands

*Time

Displays the day, date and time of day

Syntax

```
*Time
```

Parameters

None

Use

*Time displays the day, date and time of day. It is displayed in the same format as OS_Word 14,0.

Example

```
*Time
```

Related commands

None

Related SWIs

OS_Word 14,0 (SWI &07), OS_Word 15 (SWI &07),
OS_ConvertStandardDateAndTime (SWI &C0),
OS_ConvertDateAndTime (SWI &C1)

Related vectors

WordV, WrchV

18 Conversions

Introduction

This chapter is a collection of SWIs that convert from one form to another. Here is a summary of the conversions that can be done:

- Convert a number to a string in binary, decimal or hex, with some format control. You can specify that the source number in a variety of sizes: 1, 2, 3 or 4 bytes in length in most cases.
- Convert a string containing a number in any base from 2 to 36 to a number.
- Process a string with control codes and other special characters. This allows a string with any control codes to be created by passing a string with only printable characters in it.
- Substitute a string containing arguments with the given values. Used with command line arguments to an application.
- Evaluate an expression with logical, arithmetic, bit and string operations, giving a logical, numeric or string result.
- Extract options from a command line using a given key.
- Convert a SWI number to a string with its full name and vice versa.
- Convert a network station pair of numbers number into a string.
- Convert a file size into a string, for example '12 Kbytes'

Overview and Technical Details

This section leads through the details of the differing conversion calls. Whilst most are mutually independent, some SWIs may use others within this chapter to give a multi-layered functionality.

Numbers to strings

The simplest option to convert a signed 32-bit integer into a string, the most common operation, is to use OS_BinaryToDecimal (SWI &28).

For a far greater functionality, there is a set of 24 SWIs with a common calling convention that allow a wide ranging list of conversions. Generically, these SWIs are called OS_Convert*Name*Number (SWI &D0 - E8). The *name* refers to the destination format of the string. It can be hex, signed and unsigned integer (optionally with spaces between the thousands, millions and so on), or binary. The *number* is the number of bytes to use on input. For all apart from hex, this is 1, 2, 3, or 4 bytes. Hex can be 1, 2, 4, or 8 nibbles long. See the description of these SWIs for detail.

Note that OS_BinaryToDecimal is equivalent to OS_ConvertInteger4 (SWI &DC) from these SWIs.

Strings to numbers

OS_ReadUnsigned (SWI &21) will read a number in an ASCII string and convert it into an unsigned integer. The number in the string can be specified to be in any base from 2 to 36. Base 36 has 0 - 9, A - Z as numbers. No prefix means that the number is decimal by default, while the conventional '&' is used to indicate hex. All bases can be specified by the *base_number* form: eg 2_1100 is 12 in binary.

GS string operations

The GS operations are a way of putting any characters from 0 - 255 into a string using only the printable character set. OS_GSInit (SWI &25) and OS_GSRead (SWI &26) work together to scan a string on a character at a time basis. OS_GSTrans (SWI &27) performs both these functions and scans the string. Unless you need character by character control, OS_GSTrans is easier to use.

| character

The '|' character is used by OS_GSRead and OS_GSTrans as a flag for a special character. It affects how the character following it is interpreted. Here is a list of its effects:

ASCII code	Symbols used
0	|@
1 - 26	|*letter* eg |A (or |a) = ASCII 1, |M (or |m) = ASCII 13
27	|[or |{
28	|\
29	|] or |}
30	|^ or |~
31	|_ or |'
32 - 126	*keyboard character*, except for:
"	|"
|	||
<	|<
127	|?
128 - 255	|!*coded symbol* eg ASCII 128 = |!|@ ASCII 129 = |!|A

Note that '|!' means set the top bit of the following character, even if it is set by another '|' character.

To include leading spaces in a definition, the string must be in quotation marks, '"', which are not included in the definition. To include a single " character in the string, use |" or "".

Substitute arguments

The reason why '<' must be preceded by a '|' is that you can put values and variables inside angle brackets.

You can use the form <*number*>, where the number between the angle brackets will be interpreted as if it was a parameter to OS_ReadUnsigned: that is, a number in any base from 2 to 36. The value returned from this SWI will be placed as a character in the output stream; any values above bit 7 will be ignored.

A string with a name enclosed in '< >' characters will be used to look up a system variable. You must have used *Set, *SetMacro or *SetEval to set the variable. The value of the variable will be substituted for the name and the angle brackets using OS_ReadVarVal; eg if the variable 'hisname' had been set to 'Fred', then the string 'My friend's name is <hisname>' would be translated to 'My friend's name is Fred'. System variables and the calls that operate on them are described in the chapter entitled *Program Environment* on page 1-277.

Flags

There are options which can be used to determine the way in which the string is interpreted. This is done by setting the top three bits in R2 passed to OS_GSInit or OS_GSTrans, as follows:

Bit	Meaning
29	If set then a space is treated as a string terminator
30	If set control codes are not converted (ie '|' syntax is ignored)
31	Double quotation marks '"' are not to be treated specially, ie they are not stripped around strings.

*Echo

The *Echo command will pass a string through OS_GSTrans and then send it to the display.

Evaluation operators

A string containing an expression can be evaluated. An expression consists of any of the operators listed below, and strings, and numbers. It can return a result that is a number or a string. OS_EvaluateExpression (SWI &2D) is the core routine here. It is in turn called by *Eval. This allows you to perform evaluations from the command line.

The *If command also uses this call to perform a logical decision about which * Command to perform.

Any strings in the evaluation string are passed to OS_GSTrans, so all its operators will be used. This of course means that OS_ReadUnsigned and OS_ReadVarVal will in turn be called if you use a string that requires them. Note, however, that vertical bar escape sequences (eg '|G' for ASCII 7) are not recognised.

As well as passing *<name>* operators in strings to OS_ReadVarVal, any item which cannot immediately be treated as a string or a number is also looked up as a system variable. For example, in the expression FRED+1, FRED will be looked up as a variable.

The operators recognised by the expression evaluator are as follows:

Arithmetic operators

+	Add two integers
-	Subtract two integers
*	Multiply two integers
/	Integer part of division
MOD	Remainder of a division

Logical operators

=	Equal	-1 is TRUE
<>	Not equal	0 is FALSE
>=	Greater than or equal	
<=	Less than or equal	
<	Less than	
>	Greater than	

Bit operators

>>	Arithmetic shift right
>>>	Logical shift right
<<	Logical shift left
AND	AND
OR	OR
EOR	Exclusive OR
NOT	NOT

String operators

+	Concatenate two strings eg 'HI' + 'LO' = "HILO"
RIGHT *n*	Take *n* characters from the right eg "HELLO" RIGHT 2 = "LO"
LEFT *n*	Take *n* characters from the left eg "HELLO" LEFT 3 = "HEL"
LEN	Return the length of a string eg LEN "HELLO" = 5

Conversions

STR	Convert a number into a string eg STR 24 = "24"
VAL	Take the value of a string eg VAL "12d3" = 12

Where appropriate, type conversions are performed automatically. For example, if an integer is subtracted from a string, then the string is evaluated and an integer result is produced ("2"-1 gives the result 1). The null string "" is converted to 0 by both the implicit and explicit (VAL) conversions.

Similarly, integers will be converted to strings if necessary: the expression 1234 LEFT 2 will yield "12".

The operators have the same relative priorities as their equivalents in BBC BASIC, eg * is higher than + which is higher than >, etc.

Parameter substitution

Given a list of space separated arguments, OS_SubstituteArgs (SWI &43) will replace references to those parameters in a string. %0 refers to the first string in the argument list and so on. This is generally used when processing command lines.

For a more powerful handling of command lines, use OS_ReadArgs (SWI &49). This is passed a list of parameter definitions and an input string. The parameters can be described as being in any order or in a fixed order. They can handle on/off switches (ie presence is indicated), or values. The values can also be automatically passed through OS_GSTrans or OS_EvaluateExpression if required.

SWI number to string

Two calls can be used to translate a SWI number to and from its full name as a string. OS_SWINumberToString (SWI &38) will convert from a SWI number to a string, and OS_SWINumberFromString (SWI &39) will convert from a string to a SWI number.

Note that having bit 17 set will result in the string being prefixed with an 'X', and vice versa.

Econet numbers

The pair of numbers that refer to the network number and station number can be converted into a string by OS_ConvertFixedNetStation (SWI &E9). This will pad the string with leading zeros where required. If you don't want this padding, use OS_ConvertNetStation (SWI &EA).

File size

There are two SWIs that will convert a file size from an integer into a string. They can decide whether to display as bytes, Kbytes or Mbytes. OS_ConvertFileSize (SWI &EC) will convert an integer into a number up to 4 digits followed by an optional 'K' if it is in kilobytes or 'M' if in megabytes, followed by the word 'bytes' and a null to terminate.

OS_ConvertFixedFileSize (SWI &EB) is exactly the same, except that it will always print the numeric field as four characters, padding with spaces if necessary.

SWI Calls

OS_ReadUnsigned (SWI &21)

Convert a string to an unsigned number

On entry

R0 = base in the range 2 - 36 (else 10 assumed), and flags in top 3 bits
R1 = pointer to string
R2 = maximum value if R0 bit 29 set

On exit

R0 preserved
R1 = pointer to terminator character
R2 = value

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadUnsigned takes a pointer to a string and tries to convert it into an integer value which is returned in R2.

Valid strings may start with a digit (where 'digits' may also be letters, depending on the base) or one of the following:

&	The number is in hexadecimal notation
*base*_	The number is in a given base, where *base* is in the range 2 to 36. For example, 2_1010 is a base two (binary) number.

These override any base specified in R0. (If R0 contains an illegal base, 10 is assumed.) Characters following them are read until a character is reached which is not consistent with the base in use. For example, assuming R0=10 on entry, the terminator of 43AZ is A, whereas the terminator of &43AZ is Z.

In addition, R0 contains three flags which cause checks to be performed on the terminator and the range of the number obtained:

Bit	Meaning if set
31	Check terminator is a control character or space
30	Restrict value range to 0 - 255
29	Restrict range to 0 - R2 inclusive; a 'Number too big' error is given otherwise

If either of these checks fail, a 'Bad number' error is given. This error also occurs if the first character is not a valid digit. If a base is given at the start of the number and isn't in the range 2 - 36, a 'Bad base' error is given.

Related SWIs

None

Related vectors

None

OS_GSInit (SWI &25)

Initialises registers for use by OS_GSRead

On entry

R0 = pointer to string to translate
R2 = flags

On exit

R0 = value to pass back in to OS_GSRead
R1 = first non-blank character
R2 = value to pass back in to OS_GSRead

Interrupts

Interrupt state is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_GSInit is one of the string routines which are used by the operating system command line interpreter to process the strings sent to it. One of the advantages of these routines is that they enable you to use the character '|' to introduce control characters which would otherwise be difficult to enter directly from the keyboard.

See the section entitled GS *string operations* on page 1-430 for a list of the conversions that are performed by the routines, and of the flags passed in R2.

OS_GSInit also returns the first non-blank character in the string. However, this is not necessarily the same as the output from the first OS_GSRead, since OS_GSInit doesn't perform any expansion.

Related SWIs

OS_GSRead (SWI &26), OS_GSTrans (SWI &27)

Related vectors

None

OS_GSRead (SWI &26)

Returns a character from a string which has been initialised by OS_GSInit

On entry

R0 from last OS_GSRead/OS_GSInit
R2 from last OS_GSRead/OS_GSInit

On exit

R0 updated for next call to OS_GSRead
R1 = next translated character
R2 updated for next call to OS_GSRead
C flag is set if end of string reached

Interrupts

Interrupt state is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_GSRead reads a character from a string, using registers initialised by a OS_GSInit immediately prior to this call. The next expanded character is returned in R1. The values in R0 and R2 are updated so they are set up for the next call to OS_GSRead.

The interpretation of characters which pass through OS_GSRead is described in the section entitled GS *string operations* on page 1-430. Note that this call does not correctly handle quoted termination in RISC OS 2.0.

An error is returned for a bad string – for example, mismatched quotation marks.

Related SWIs

OS_GSInit (SWI &25), OS_GSTrans (SWI &27)

Related vectors

None

OS_GSTrans (SWI &27)

Equivalent to a call to OS_GSInit and repeated calls to OS_GSRead

On entry

R0 = pointer to string, terminated by ASCII 10 (LF) or 13 (CR) or 0 (NUL)
R1 = buffer pointer
R2 = buffer size (*maxlen*) and flags in top 3 bits

On exit

R0 = pointer to character after terminator
R1 = pointer to buffer, or 0
R2 = number of characters in buffer, or *maxlen*+1 if the buffer overflowed
C flag is set if buffer overflowed

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_GSTrans is equivalent to a call to OS_GSInit followed by repeated calls to OS_GSRead until the end of the source string is reached. Each time it obtains a character and translates it, OS_GSTrans then places it in a buffer.

The flags in R2, on entry, are the same as those supplied to OS_GSInit. On exit, R0 points to the character after the terminator of the source string, and R1+R2 points to the terminator of the translated string. If the C flag is set on exit the buffer was too small for the translated string; R2 is set to the length of the buffer plus one.

The flags and interpretation of characters which pass through OS_GSTrans are described in the section entitled GS *string operations* on page 1-430. Note that this call does not correctly handle quoted termination in RISC OS 2.0.

An error is returned for a bad string – for example, mismatched quotation marks.

Related SWIs

OS_GSInit (SWI &25), OS_GSRead (SWI &26)

Related vectors

None

OS_BinaryToDecimal (SWI &28)

Convert a signed number to a string

On entry

R0 = signed 32-bit integer
R1 = pointer to buffer
R2 = maximum length

On exit

R0, R1 preserved
R2 = number of characters given

Interrupts

Interrupt state is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_BinaryToDecimal takes a signed 32-bit integer in R0 and converts it to a string, placing it in the buffer. R1 points to the buffer and R2 contains its maximum length. Leading zeros are suppressed and the string will start with a minus sign, '-', if R0 was negative. The number of characters given is returned in R2.

The error 'Buffer overflow' is given if the converted string is too long to fit in the buffer.

Related SWIs

None

Related vectors

None

OS_EvaluateExpression (SWI &2D)

Evaluate a string expression and return an integer or string result

On entry

R0 = pointer to string
R1 = pointer to buffer
R2 = length of buffer

On exit

R0 preserved
R1 = 0 if an integer returned, else preserved
R2 = integer result, or length of string in buffer

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_EvaluateExpression takes a string pointed to by R0, evaluates it, and places the result in the buffer which is pointed to by R1. Its maximum length is R2. The type of the result is given by R1 as follows:

Value	Meaning
0	Integer result returned in R2
Not 0	String is returned in buffer, length returned in R2, R0 and R1 preserved

See the section entitled *Evaluation operators* on page 1-432 for a description of the operators that you can use. Note that monadic plus/minus operators are not correctly handled in RISC OS 2.0 (eg *Eval 50*-3 gives a 'Missing operand' error).

If the buffer is not large enough to hold the resulting string, then a 'Buffer overflow' error is generated.

Related SWIs

None

Related vectors

None

OS_SWINumberToString (SWI &38)

Convert a SWI number to a string containing its name

On entry

R0 = SWI number
R1 = pointer to buffer
R2 = buffer length

On exit

R0, R1 preserved
R2 = length of string in buffer

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_SWINumberToString converts a SWI number to a SWI name.

The returned string is null-terminated, and starts with an X if the SWI number has bit 17 set.

SWI numbers < &200 have an 'OS_' prefix to the main part, and a SWI-dependent end section (which is 'Undefined' for unknown OS SWIs).

SWI numbers in the range &100 to &1FF are converted in the form OS_WriteI+"A", or OS_WriteI+23 if the character is not a printable one.

SWI numbers &200 are looked for in modules. If a suitable name is found, it is given in the form *module_name* or *module_number*, eg. Wimp_Initialise, Wimp_32. If no name is found in the modules, the string 'User' is returned.

Note that this call does not correctly handle negative SWI numbers in RISC OS 2.0.

Related SWIs

OS_SWINumberFromString (SWI &39)

Related vectors

None

OS_SWINumberFromString (SWI &39)

Convert a string to a SWI number if valid

On entry

R1 = pointer to name (which is terminated by a character ≤ 32)

On exit

R0 = SWI number
R1 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_SWINumberFromString converts a SWI name to a SWI number. An error is given if the SWI name is not recognized.

The conversion is as follows:

- A leading X is checked for and stripped. If present, &20000 is added to the number returned (ie bit 17 will be set).
- System names are checked for. Note that the conversion of SWIs is not quite bidirectional: the name OS_Writel+" " can be produced, but only OS_Writel is recognized.
- Modules are scanned. If the module prefix matches the one given, and the suffix to the name is a number, then that number is added to the module's SWI 'chunk' base, and the sum returned. For example, Wimp_&23 returns &400E3, as the Wimp's chunk number is &400C0.

- If the suffix is a name, and this can be matched by the module, the appropriate number is returned. For example, Wimp_Poll returns &400C7.

See the chapter entitled *Modules* on page 1-191 for more information on how modules provide the conversion.

Note that SWI names are case sensitive, so you must spell them exactly as returned by OS_SWINumberToString.

Related SWIs

OS_SWINumberToString (SWI &38)

Related vectors

None

OS_SubstituteArgs (SWI &43)

Substitute command line arguments

On entry

R0 = pointer to argument list, and flag in top bit
R1 = pointer to buffer for result string
R2 = length of buffer
R3 = pointer to template string
R4 = length of template string

On exit

R0, R1 preserved
R2 = number of characters in result string (including the terminator)
R3, R4 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call performs the hard work involved in substituting a list of arguments into a 'template' string. Its main use is in the processing of command AliasS... variables by the system. As it is also useful in other situations, it has been made available to users. For example, FileSwitch uses it in the processing of AliasS@LoadType_TTT variables.

The argument list is a string consisting of space-separated items which will be substituted into the template string. Spaces within double quotation marks are not counted as argument separators. Typically, the argument string will just be the tail of a * Command. It is control-character terminated.

The result of substituting the arguments into the template string is placed in the buffer. The length of the buffer is given so that the call can check for buffer overflow.

The template string is copied into the result buffer character for character. However, when a '%' appears in the template string (even within quotation marks), it marks where an argument should be placed into the output buffer. The '%' is followed by a single digit from 0 to 9. %0 stands for the first argument in the argument list, and so on. %*n means all of the arguments from the *n*th one onwards. %% means a single '%'. Anything else following the '%' is not treated specially, ie both the '%' and the character are copied over.

The template string does not have a terminator; instead its length is given. At the end of the substitution, any arguments after the highest one mentioned in the template string are appended to the result string. This can be stopped by setting the top bit of R0 on entry.

If a non-existent argument is specified in the template string, then the substitution process is terminated. No error is given.

Related SWIs

None

Related vectors

None

OS_ReadArgs (SWI &49)

Given a keyword definition, scan a command string

On entry

R0 = pointer to keyword definition
R1 = pointer to input string
R2 = pointer to output buffer
R3 = size of output buffer

On exit

R0 - R2 preserved
R3 = bytes left in output buffer

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI processes a command string using a keyword definition for syntax. The results are written out to the output buffer using a specialised format for this command.

Keyword definition

The keyword definition defines the parameters that can be in the command string. It is composed of a sequence of keywords, separated by commas. Each of these is made up of one or two names, followed by a sequence of qualifiers. The syntax of a keyword is:

[*keyword_name*[=*alias_name*]][/*qualifier*...]

The *keyword_name* is what you want users to identify the parameter with. This can be any string composed of alphanumerics and the '_' character. The alias name is an optional alternative name for the same keyword. You can have a keyword with no name. See the command string description below of how to set it.

The qualifier describes what kind of a parameter it is. There can be as many qualifiers as you like with one parameter, but some are mutually exclusive. The qualifiers can be any one of the following characters in upper or lower case:

- /A keyword must always be given a value
- /K keyword must always precede its value
- /S the option is a switch ie presence only is reported
- /E OS_EvaluateExpression will be called to transform the value. This can return a number or a string. Note that numeric evaluations only can be performed.

  Note that in RISC OS 2.0 a 'Buffer full' error is generated if this argument evaluates to a string
- /G OS_GSTrans will be called to transform the value

Command string

The command string contains a sequence of commands using the syntax defined by the keyword definition. A command string is made of definitions of the following syntax:

`[-keyword_name] value`

If the keyword name is used, then the value will be attached to the named keyword. These can appear in any arbitrary order in the command string. The name after the '-' can be the full name of the keyword or its alias, or the first letter of either. For example, if the keyword definition contains "name=title", then all of the following are valid in the command string:

"-name fred", "-title fred", "-n fred", "-t fred"

Note that if more than one keyword has the same first letter, then the single letter form will be used by the first occurrence of a given letter in the keyword definition.

Also note that case is ignored, so "-FILE" and "-file" are identical.

If a definition has no *-keyword_name* preceding it, then the first unused keyword that is not a switch in the definition string will be given that value. This is how nameless keywords are set. For example, if the definition string is "infile/a,outfile" and the command string is "-infile one -outfile two three", then the first and nameless keyword will be set to three, because it was the first undefined keyword in the definition.

Keywords are marked by a preceding '-' character, but this does not disallow these characters from appearing in values anywhere but at the start. For example, if the keyword definition is "formula/e", then "-formula 6-3" will set it to the value of 3. If the command is "-formula -3+6", then this will cause an error.

Whilst some evaluated expressions can be done without spaces (1+2 for example), there are many that cannot. You can evaluate an expression in quotes, which allow spaces, as in this example:

```
"&3F AND &17"
```

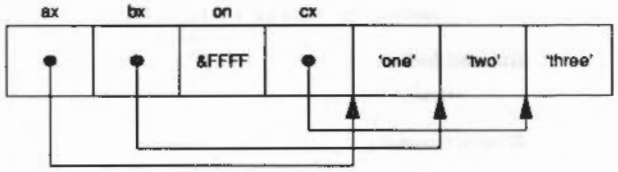
With GSTrans'd strings, if you want to put a quoted string inside quotes then you must use double quotes, as follows:

```
"This is ""IT"""
```

Output buffer

The output buffer contains the results for all of the possible keywords. For *n* keywords in the keyword definition, the first *n* words of the output buffer contain the results of the parsing of the command line. If the keyword was a switch (with /S qualifier), then a non-zero value indicates that the switch was used. For all other kinds of result, then it is a pointer; a pointer of zero indicates that the parameter was not present. These results are appended sequentially to the output buffer.

The following example uses a keyword definition of "ax,bx,on/s,cx" and a command string of "one two three -on". The output buffer looks like this:



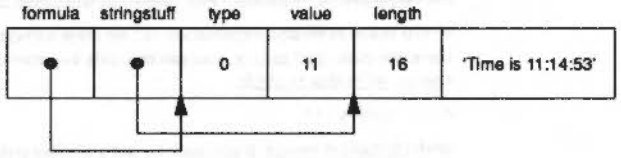
The results of GSTransed strings and evaluated expressions are stored differently. In a GSTransed string, the result pointer points to a block of the following format:

- *length* two byte length
- *string* *length* bytes of string

In an evaluated expression, the pointer points to a block like the following:

- *type* one byte result type (which at present can only be zero or an integer)
- *value* four byte integer

For an example showing /e and /g switches, if the keyword definition was "formula/e,time/g" and the command string was "-f 6+6-1 -t ""Time is <SysSTime>""", then the result looks like this:



Examples

Keyword definition:

```
number=times/e,file/k/a,expandtabs/s
```

can be matched by any of:

```
-n 10 -file jeff
-times 1+7 -file jeff -expandtabs
-file thingy -e
```

but not by either of:

```
thingy -number 4
-number 20 -times 4 -file jeff
```

Related SWIs

None

Related vectors

None

OS_Convert*NameNumber* (SWIs &D0 - E8)

These calls convert a number into a string

On entry

R0 = value to be converted
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWIs are re-entrant

Use

This range of SWIs use a common form and can convert a number into a string in a variety of ways.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

The *name* part of the SWI name can be any of the following groups:

Hex

Convert to a hexadecimal string

The *number* is the number of ASCII digits in the output string, either 1, 2, 4, 6 or 8. Only enough significant bits to perform the conversion are used. The string does not include an ampersand ('&') but is padded with leading zeros, so is of fixed length. The SWIs in this group are:

SWI name	SWI number	Output for... zero	largest value
OS_ConvertHex1	&D0	'0'	'F'
OS_ConvertHex2	&D1	'00'	'FF'
OS_ConvertHex4	&D2	'0000'	'FFFF'
OS_ConvertHex6	&D3	'000000'	'FFFFFF'
OS_ConvertHex8	&D4	'00000000'	'FFFFFFFF'

Cardinal

Convert to an unsigned decimal number

The *number* is the number of bytes to be used from the input value. The string is not padded with zeros, so is of variable length. The SWIs in this group are:

SWI name	SWI number	Output for... zero	largest value
OS_ConvertCardinal1	&D5	'0'	'255'
OS_ConvertCardinal2	&D6	'0'	'65535'
OS_ConvertCardinal3	&D7	'0'	'16777215'
OS_ConvertCardinal4	&D8	'0'	'4294967295'

Integer

Convert to a signed decimal number

The *number* is the number of bytes to be used from the input value. If the most significant bit is set (of the *number* bytes used), the number is taken to be negative, and a leading '-' is produced. The string is not padded with zeros, so is of variable length. The SWIs in this group are:

SWI name	SWI number	Output for... largest -ve	largest +ve value
OS_ConvertInteger1	&D9	'-128'	'127'
OS_ConvertInteger2	&DA	'-32768'	'32767'
OS_ConvertInteger3	&DB	'-8388608'	'8388607'
OS_ConvertInteger4	&DC	'-2147483648'	'2147483647'

Binary

Convert to a binary number

The *number* is the number of bytes to be used from the input value. The string is padded with leading zeros, so is of fixed length (*number* x 8). The SWIs used in this group are:

SWI name	SWI number	Output for largest value
OS_ConvertBinary1	&DD	'11111111'
OS_ConvertBinary2	&DE	'1111111111111111'
OS_ConvertBinary3	&DF	'111111111111111111111111'
OS_ConvertBinary4	&E0	'11111111111111111111111111111111'

SpacedCardinal

Convert to an unsigned decimal number, with spaces every three digits

The *number* is the number of bytes to be used from the input value. The string is not padded with zeros, so is of variable length. In addition, every three digits from the right, a space is inserted. The SWIs used in this group are:

SWI name	SWI number	Output for... zero	largest value
OS_ConvertSpacedCardinal1	&E1	'0'	'255'
OS_ConvertSpacedCardinal2	&E2	'0'	'65 535'
OS_ConvertSpacedCardinal3	&E3	'0'	'16 777 215'
OS_ConvertSpacedCardinal4	&E4	'0'	'4 294 967 295'

SpacedInteger

Convert to a signed decimal number, with spaces every three digits

The *number* is the number of bytes to be used from the input value. If the most significant bit is set (of the *number* bytes used), the number is taken to be negative, and a leading '-' is produced. The string is not padded with zeros, so is of variable length. In addition, every three digits from the right, a space is inserted. The SWIs in this group are:

SWI name	SWI no.	Output for... largest -ve	largest +ve val.
OS_ConvertSpacedInteger1	&D9	'-128'	'127'
OS_ConvertSpacedInteger2	&DA	'-32 768'	'32 767'
OS_ConvertSpacedInteger3	&DB	'-8 388 608'	'8 388 607'
OS_ConvertSpacedInteger4	&DC	'-2 147 483 648'	'2 147 483 647'

Related SWIs

OS_BinaryToDecimal (SWI &28)

Related vectors

None

OS_ConvertFixedNetStation (SWI &E9)

Convert from an Econet station/network number pair to a string

On entry

R0 = pointer to two word block (value to be converted)
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null zero in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

R0 points to two words in memory. The first word contains the station number and the second word contains the network number.

This call always converts into a form *nnn.sss*, where *nnn* is the network number and *sss* the station number. If the network number is zero, the first four characters are spaces; if it is non-zero, leading zeros are converted to spaces. If the network number was zero, leading zeros in the station number are converted to spaces; otherwise they are left as zeros.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertNetStation (SWI &EA)

Related vectors

None

OS_ConvertNetStation (SWI &EA)

Convert from an Econet station/network number pair to a string

On entry

R0 = pointer to two word block (value to be converted)
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

R0 points to two words in memory. The first word contains the station number and the second word contains the network number.

This call performs the same conversion as OS_ConvertFixedNetStation, but suppresses zeros and spaces wherever possible, to yield the shortest possible string.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertFixedNetStation (SWI &E9)

Related vectors

None

OS_ConvertFixedFileSize (SWI &EB)

Convert an integer into a filesize string of a fixed length

On entry

R0 = filesize in bytes
R1 = pointer to buffer
R2 = length of buffer in bytes

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI will convert an integer into a filesize string of a fixed length. The format of the string is:

4_digit_number `M|K|`*space*`bytes`*null*

The *4_digit_number* at the start is padded with spaces if it is less than four digits in length; *space* and *null* are the ASCII characters 32 and 0 respectively.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertFileSize (SWI &EC)

Related vectors

None

OS_ConvertFileSize (SWI &EC)

Convert an integer into a filesize string

On entry

R0 = filesize in bytes
R1 = pointer to buffer
R2 = length of buffer in bytes

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI will convert an integer into a filesize string. The format of the string is:

number `M|K|` *space* `bytes` *null*

The *number* at the start is up to four digits in length; *space* and *null* are the ASCII characters 32 and 0 respectively.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertFixedFileSize (SWI &ED)

Related vectors

None

*Commands

*Echo

Displays a string on the screen (after translating it using OS_GSTrans)

Syntax

```
*Echo string
```

Parameters

string	string to display

Use

*Echo takes the string following it, translates it using OS_GSTrans, and then displays it on the screen.

The main use for *Echo is in command scripts, where the command provides a useful way of checking the progress of a script, especially when debugging a faulty script, or when monitoring the progress of a series of operations.

Example

```
*Echo |GError!|M
```

Related commands

None

Related SWIs

None

Related vectors

None

*Eval

Evaluates an integer, logical, bit or string expression

Syntax

```
*Eval expression
```

Parameters

expression	any combination of the operators listed earlier

Use

*Eval evaluates an integer, logical, bit or string expression, carrying out type conversions where necessary, in a similar way to the BASIC EVAL command. It will not handle floating point numbers. You can use *Eval to do simple arithmetic (although the desktop Calculator is easier to use for four-function arithmetic), or to evaluate more complex expressions. Programmers may find the command useful for doing 'off-line' calculations (checking on remaining space, for example).

See the section entitled *Evaluation operators* on page 1-432 for a description of the operators that you can use. Note that monadic plus/minus operators are not correctly handled in RISC OS 2.0 (eg *Eval 50*-3 gives a 'Missing operand' error).

Example

```
*Eval 127 * 23 >> 2
Result is an integer, value : 730
```

Related commands

*If, *SetEval

Related SWIs

OS_EvaluateExpression (SWI &2D)

Related vectors

None

*If

Conditionally executes a * Command, depending on the value of an expression

Syntax

```
*If expression Then command [Else command]
```

Parameters

expression	an integer expression
command	any valid * Command

Use

*If conditionally executes a * Command, depending on the value of an expression. The expression can be any integer expression, including (if necessary) variable names enclosed in angled brackets.

The expression is evaluated by the operating system's expression evaluation. If the If-expression evaluates to a non-zero value, the Then-clause is executed. If the If-expression evaluates to zero, and there is an Else-clause, the Else-clause is evaluated.

If you wish to compare a variable to a string both must be enclosed in double quotes to ensure a string comparison is performed; see the first example.

See the section entitled *Evaluation operators* on page 1-432 for a description of the operators that you can use.

Example

```
*If "<name>" = "Michael" Then echo Hi Mike! Else Echo Go away <name>!

If <Sys$Year>=1988 Then Run Calendar
```

Related commands

*Eval

Related SWIs

None

Related vectors

None

19 Extension ROMs

Extension ROMs are ROMs fitted in addition to the main ROM set, which provide software modules which are automatically loaded by RISC OS on a reset. Note that **RISC OS 2.0 does not support extension ROMs**.

Using extension ROMs, you can add extra modules to RISC OS, or provide replacement modules for those already in RISC OS.

It is the Expansion Card Manager's responsibility to recognise extension ROMs. For it to do so, your extension ROMs need to have headers, which are detailed in the chapter entitled *Expansion Cards and Extension* ROMS on page 6-85. This chapter also gives details of the software that RISC OS provides to manage and communicate with extension ROMs (and, of course, expansion cards). Expansion cards and extension ROMs are covered together because both use substantially the same layout of code and data, and the same SWIs.

It is the kernel's responsibility to load any relocatable modules from any extension ROMs – once the Expansion Card Manager has recognised them. The chapter entitled *Modules* on page 1-191 gives details of how the modules are loaded on a reset. It also tells you how to write the relocatable modules held in extension ROMs.