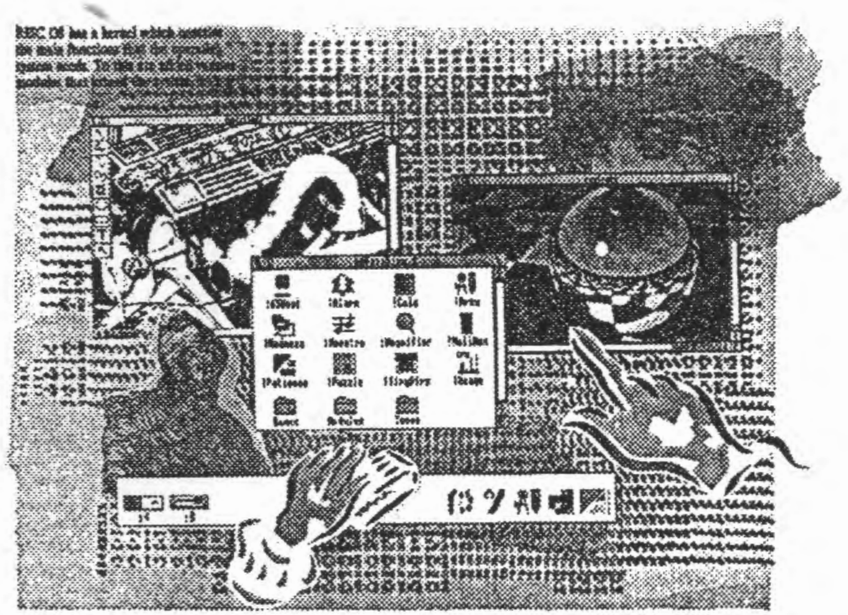
RISC OS

PROGRAMMER'S REFERENCE MANUAL

Volume II



Acorn

Published by Acorn Computers Limited
ISBN 1 85250 112 4
Edition 1
Part number 0470,292
Issue 1, October 1991

Contents

About this manual 1-ix

Part 1 – Introduction 1-1

An introduction to RISC OS 1-3
ARM Hardware 1-7
An introduction to SWIs 1-21
* Commands and the CLI 1-31
Generating and handling errors 1-37
OS_Byte 1-45
OS_Word 1-55
Software vectors 1-59
Hardware vectors 1-103
Interrupts and handling them 1-109
Events 1-137
Buffers 1-153
Communications within RISC OS 1-167

Part 2 – The kernel 1-189

Modules 1-191
Program Environment 1-277
Memory Management 1-329
Time and Date 1-391
Conversions 1-429
Extension ROMs 1-473
Character Output 2-1
VDU Drivers 2-39
Sprites 2-247
Character Input 2-337
The CLI 2-429
The rest of the kernel 2-441

Part 3 – Filing systems 3-1
Introduction to filing systems 3-3
FileSwitch 3-9
FileCore 3-187
ADFS 3-251
RamFS 3-297
DOSFS 3-305
NetFS 3-323
NetPrint 3-367
PipeFS 3-385
ResourceFS 3-387
DeskFS 3-399
DeviceFS 3-401
Serial device 3-419
Parallel device 3-457
System devices 3-461
The Filer 3-465
Filer_Action 3-479
Free 3-487
Writing a filing system 4-1
Writing a FileCore module 4-63
Writing a device driver 4-71

Part 4 – The Window manager 4-81
The Window Manager 4-83
Pinboard 4-343
The Filter Manager 4-349
The TaskManager module 4-357
TaskWindow 4-363
ShellCLI 4-373
!Configure 4-377

Part 5 – System extensions 4-379
ColourTrans 4-381
The Font Manager 5-1
Draw module 5-111
Printer Drivers 5-141
MessageTrans 5-233
International module 5-253
The Territory Manager 5-277
The Sound system 5-335
WaveSynth 5-405
The Buffer Manager 5-407
Squash 5-423
ScreenBlank 5-429
Econet 6-1
The Broadcast Loader 6-67
BBC Econet 6-69
Hourglass 6-73
NetStatus 6-83
Expansion Cards and Extension ROMS 6-85
Debugger 6-133
Floating point emulator 6-151
ARM3 Support 6-173
The Shared C Library 6-183
BASIC and BASICTrans 6-277
Command scripts 6-285

Appendices and tables 6-293
Appendix A: ARM assembler 6-295
Appendix B: Warnings on the use of ARM assembler 6-315
Appendix C: ARM procedure call standard 6-329
Appendix D: Code file formats 6-347
Appendix E: File formats 6-387
Appendix F: System variables 6-425
Appendix G: The Acorn Terminal Interface Protocol 6-431
Appendix H: Registering names 6-473
Table A: VDU codes 6-481
Table B: Modes 6-483
Table C: File types 6-487
Table D: Character sets 6-491

Indices Indices-1

Index of * Commands Indices-3
Index of OS_Bytes Indices-9
Index of OS_Words Indices-13
Numeric index of SWIs Indices-15
Alphabetic index of SWIs Indices-27
Index by subject Indices-37

20 Character Output

Introduction

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

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

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

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

Overview

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

Terminology used

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

Back-doors

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

Device independence

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

OS_WriteC

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

Buffers

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

Devices

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

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

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

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

Printer stream

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

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

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

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

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

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

The network printer is the one that is accessed remotely across a network. See the chapter entitled *NetPrint* on page 3-367 for details of this.

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

Note that this chapter concerns itself only with the character print routines. See the chapter entitled *Printer Drivers* on page 5-141 for information on the drivers that must be used for any graphical printing.

Serial output device

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

The serial hardware itself changes the character into a series of voltage changes on its connection with the outside. These voltages and other control lines work together to communicate with another serial port on another machine. The baud rate of a serial port is the number of bits per second that it is sending or receiving. Under RISC OS, these rates can be controlled independently, although not all machines will support different transmit and receive rates.

Calls that are specific to the serial port, whether they refer to input or output (eg those to set the baud rate, or to explicitly send/receive a character from/to the serial port), are gathered together in the chapter entitled *Serial device* on page 3-419.

Spool device

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

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

VDU device

The VDU device driver will put any characters or graphics onto the screen. Some characters are displayed directly, while others are interpreted as graphics commands. This chapter contains details of the interface to the VDU system, but for a detailed description of the VDU system, refer to the chapter entitled *VDU Drivers* on page 2-39.

Technical Details

Device independence

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

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

Printer stream

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

OS_Byte 3

Three bits in the byte sent to OS_Byte 3 to select output streams control whether a character is sent to the printer. In addition, a character may also be sent to the printer under the control of the VDU stream.

Bit 2 provides global control over the printer. If this bit is set, then it is not possible for OS_WriteC to cause a character to be inserted into the printer buffer. If it is clear, then the character may or may not be sent to the printer, depending on the state of the other bits.

Bit 6 acts in a similar way: if it is clear, characters may be sent to the printer, but if it is set, they are stopped. There is still one way of getting characters to the printer if bit 6 is set; this is described below.

Assuming bits 2 and 6 are clear, then the simplest way of enabling the printer is by setting bit 3. When this is done, all characters sent to OS_WriteC (except the printer ignore character) will be inserted into the printer buffer.

VDU printer control

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

done, all printable characters and some control characters sent to the VDU stream will also go to the printer. Sending ASCII 3 (Ctrl-C) to the VDU disables the copying of characters to the printer.

A further control code, ASCII 1 (Ctrl-A), causes the next character to be sent to the printer (if enabled by Ctrl-B), but not to the screen. All characters may be sent this way, including the control codes which are usually ignored by the VDU printer stream, and the printer ignore character.

If either bit 6 or bit 2 of the streams byte is set, then the VDU printer stream has no effect. The exception is when the character is preceded by a Ctrl-A. In this case, bit 6 will not prevent the character from being sent, although bit 2 will.

More details of the VDU printer stream control codes are given in the chapter entitled VDU Drivers on page 2-39.

The flow of control is summarised by the diagram below:

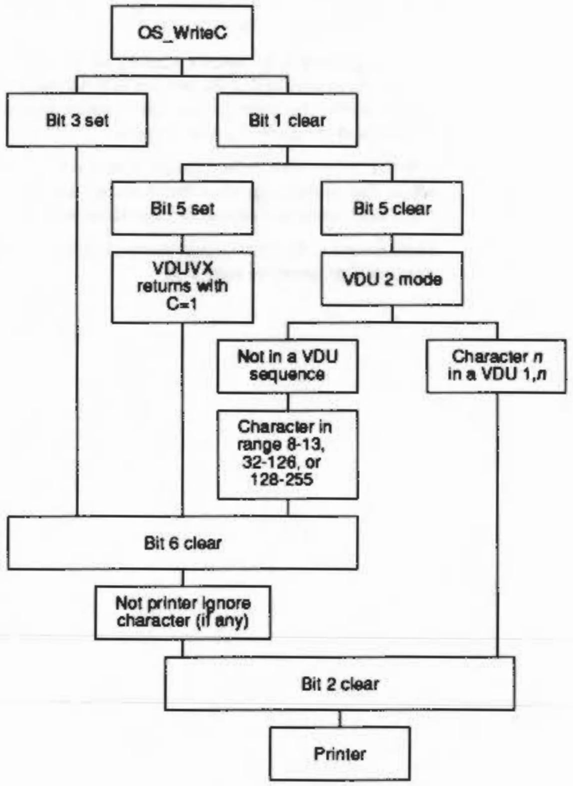


Figure 20.1 *Flow of control in the printer stream*

OS_Byte 5

Regardless of how a character gets to the printer stream, it is then sent to the current printer device. This is set by OS_Byte 5. It is passed a byte which can select one of 256 potential drivers, 4 of which are supplied with RISC OS.

- printer sink
- parallel
- serial
- network

When an OS_Byte 5 is used, the new destination streams come into effect only when all the current contents of the printer buffer have been sent to the previously-selected driver. This means that when you issue this OS_Byte, the calling task may appear to hang until the current printer buffer's contents are cleared. This may be forced by generating an escape condition.

The default printer device is stored in CMOS RAM and is set by *Configure Print.

OS_Byte 245

OS_Byte 245 (SWI &F5) may be used to read the current printer type, but not to set it, as it does not wait for the printer buffer to empty first. Because of this, it does not enable interrupts, so may be used to read the printer type from within an interrupt routine.

Ignore character

The printer ignore character is one which is suppressed from the printer stream, unless it got there via the VDU printer stream and was preceded by ASCII 1 (Ctrl-A). The character can be set and read using OS_Byte 246. For compatibility with older Acorn operating systems, OS_Byte 6 can also set it and OS_Byte 245 can read it.

*Ignore can be used to set the printer ignore character from the CLI. *Configure Ignore will set it permanently in CMOS RAM. The default value is 10, an ASCII linefeed.

No Ignore

There may be no printer ignore character, in which case all characters are sent. This is called the *NoIgnore* state and can be set with OS_Byte 182.

*Ignore with no parameter has the same effect from the CLI. *Configure Ignore will set the NoIgnore state permanently in CMOS RAM.

Serial device

The serial device is provided as a DeviceFS (*Device Filing System*) device. For full details, see the chapter entitled *DeviceFS* on page 3-401, and the chapter entitled *Serial device* on page 3-419. The latter chapter also contains all calls that are specific to the serial port, whether they refer to input or output (eg those to set the baud rate, or to explicitly send/receive a character from/to the serial port).

OS_Byte calls in this chapter

Sending a byte

OS_Bytes 3 and 5 can be used to select the serial port as an output stream. OS_WriteC and the SWIs that use it would be used to write to its buffer, with RISC OS handling buffer full conditions and so on. (However, there are preferred calls for sending a byte to the serial port; see the chapter entitled *Serial device* on page 3-419.)

When bit 0 of the OS_Byte 3 streams byte is set, characters sent to OS_WriteC are passed to the serial output stream. In particular, they are inserted into the serial output buffer (buffer number 2), where they remain until removed by the interrupt routine dealing with serial transmission.

Note that if the serial port is selected as the printer by OS_Byte 5, and the serial port is enabled by setting bit 0 of the stream's byte with OS_Byte 3, then the character is inserted into both buffers. This means that eventually the character is printed twice, first from the serial output buffer and then from the printer buffer. To solve this problem, make the printer another device type, such as the printer sink, which allows data sent to the printer to be ignored.

Spool device

When a spool file is opened, all characters subsequently displayed using OS_WriteC are also sent to that file, using the OS_BPut routine. This action continues until the file is closed.

Opening and closing

There are two ways of opening and closing a spool file. The simplest is to use the CLI commands *Spool or *SpoolOn to start output going into the named file.

To stop spooling and close the file, a *Spool or *SpoolOn command with no parameters must be issued, or you can stop it directly by using OS_Byte 199 documented below.

OS_Byte 3

The spool file stream can be temporarily disabled by setting bit 4 of the streams byte in OS_Byte 3. This does not close the file, but prevents OS_WriteC from trying to send the character to file.

OS_Byte 199

OS_Byte 199 (SWI &C7) provides direct control over the spool file, without the necessity of using the CLI. It reads and writes the location which holds the handle of the current spool file. If this is zero, OS_WriteC makes no attempt to use the spool stream, as no file is open. You will only need to use this command for sophisticated programs that, say, keep swapping between several spool files.

VDU device

The VDU driver will display characters and graphics on the screen. The value of the character sent determines its effect. Below is a list of the meanings of different characters. Note that in Teletext modes, a different set is in use.

Character	Meaning
0 - 31	VDU commands (graphics and control)
32 - 126	ASCII characters
127	Delete
128 - 159	User definable characters
160 - 255	ISO international characters

Note that if defining characters in the range 128 - 159 under the Desktop, you should always first read the current definition of the character using OS_Word 10 and then redefine it for the duration of the redraw. Always ensure that the character definition is restored (**not** set to the default using *FX 25) before calling XWimp_Poll again.

Disabling VDU driver

If an OS_Byte 3 with bit 1 set is sent, then the VDU driver is disabled. This prevents all output from appearing on the screen. Also, as control codes will not be acted on, it disables the VDU printer stream, described in an earlier section.

Disabling the VDU, by setting this bit, is independent of the ASCII 21 (Ctrl-U), which will disable the VDU drivers. The main difference is that the VDU printer stream will still work, if already enabled by ASCII 2 (Ctrl-B), after an ASCII 21.

VDUXV

VDUXV is the VDU extension vector. When an OS_Byte 3 with bit 1 clear (VDU enabled) and bit 5 set (VDUXV enabled) is issued, characters that would usually be sent to the VDU drivers are sent instead to the routine on the VDU extension vector. This allows you to replace the VDU drivers, usually temporarily. The font manager, for example, uses this facility.

The character sent to VDUXV can be sent to the printer stream by setting the carry flag on return from the vector.

See the chapter entitled *Software vectors* on page 1-59 for more details on installing a routine on this vector.

Direct Control

OS_Plot can be used to write to the VDU directly rather than going through the stream system. It is consequently faster. It is described on page 2-225.

SWI Calls

OS_WriteC (SWI &00)

Writes a character to all of the active output streams

On entry

R0 = character to write

On exit

R0 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

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

OS_WriteC calls the Write character vector WrchV, the default action of which is to send the character to all active output streams. If this vector is replaced using OS_Claim (see page 1-61), then all of the SWIs that use this vector will be funnelled into the replacement routine.

All the routines that call OS_WriteC may not actually call OS_WriteC or even WrchV unless there is some pressing reason to do so. For example, if WrchV is being intercepted by someone else as well as the default ROM routine, or if a spool file is active, or if the printer is active etc.

Related SWIs

OS_WriteS (page 2-15), OS_Write0 (page 2-16), OS_NewLine (page 2-17), OS_PrettyPrint (page 2-30), OS_WriteN (page 2-34), OS_WriteI (page 2-35), OS_Byte 3 (page 2-18)

Related vectors

WrchV

OS_WriteS (SWI &01)

Writes the following string to all of the active output streams

On entry

—

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sends the string that immediately follows the SWI instruction to all of the active output streams. It uses OS_WriteC directly a character at a time. The string is terminated by a null byte.

This SWI alters its return address so that execution continues at the word after the end of the string. Consequently you must not conditionally execute this SWI.

Related SWIs

OS_WriteC (page 2-13)

Related vectors

WrchV

OS_Write0 (SWI &02)

Writes an indirect string to all of the active output streams

On entry

R0 = pointer to null-terminated string to write

On exit

R0 = pointer to the byte after the null byte

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call sends the string pointed to by R0 to all of the active output streams. It uses OS_WriteC directly a character at a time. The string is terminated by a null byte.

Related SWIs

OS_WriteC (page 2-13)

Related vectors

WrchV

OS_NewLine (SWI &03)

Writes a line feed followed by a carriage return to all of the active output streams.

On entry

—

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call writes a line feed followed by a carriage return to all of the active output streams. It uses two calls to OS_WriteI to do so, which in turn call OS_WriteC.

Related SWIs

OS_WriteC (page 2-13), OS_WriteI (page 2-35)

Related vectors

WrchV

OS_Byte 3 (SWI &06)

Selects the output streams that are active

On entry

R0 = 3 (reason code)
R1 = bit mask for output streams

On exit

R0 preserved
R1 = previous stream specification
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call selects which output stream(s) are active, and will hence receive all subsequent output. A bit mask in R1 determines this:

Bit	Effect if set
0	Enables serial driver
1	Disables VDU drivers
2	Disables VDU printer stream
3	Enables printer (independently of the VDU)
4	Disables spooled output
5	Calls VDUXV instead of VDU drivers (see the chapter on VDU)
6	Disables printer, apart from VDU 1,n
7	Not used

The interpretations of all of these bits are described in subsequent sections. All bits are zero by default. This means that the VDU drivers, the VDU printer stream and the spool stream are enabled, and other streams disabled

Details of how bits 1, 2, 3 and 6 interact is described in the section entitled *Technical Details* on page 2-6 onwards.

Related SWIs

OS_Byte 236 (page 2-26)

Related vectors

ByteV, VDUXV, WrchV

OS_Byte 5 (SWI &06)

Sets the printer driver type

On entry

R0 = 5 (reason code)
R1 = printer driver type

On exit

R0 = preserved
R1 = previous printer driver type
R2 = corrupted

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets which printer driver type (and hence printer port) is used for subsequent printer output. The value of R1 on entry determines this:

Value of R1	Printer driver type
0	Printer sink
1	Parallel (Centronics) printer driver
2	Serial output
3 - 255	Files in system variables PrinterType$*n* (eg the NetPrint module sets up PrinterType$4)

The default state is set by *Configure Print.

Note that if the serial port is selected as the printer, and the serial port is enabled by setting bit 0 of the stream's byte, then the character is inserted into both buffers. This means that eventually the character is printed twice (first from the serial output buffer), so this practice is not recommended.

Instead of choosing an physical driver type, for example a parallel printer driver, you may select a 'printer sink'. This means that all characters sent to the printer are ignored.

The new destination type comes into effect only when all the current contents of the printer buffer have been sent to the previously selected driver. This means that when this OS_Byte is issued, or the corresponding *FX command, the machine may appear to hang until the current printer buffer's contents are cleared. (You may force this to happen by acknowledging an escape condition from the foreground, provided that the escape side effects are enabled.)

Related SWIs

OS_Byte 8 (page 3-426), OS_Byte 245 (page 2-27)

Related vectors

ByteV

OS_Byte 6 (SWI &06)

Sets the printer ignore character

On entry

R0 = 6 (reason code)
R1 = ASCII code of ignore character

On exit

R0 = preserved
R1 = previous ignore character
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the printer ignore character to the specified ASCII code. This character is filtered out when printing is enabled via the VDU printer stream or OS_Byte 5(page 2-20).

The default value of the printer ignore character is set by *Configure Ignore. You may temporarily change it using this OS_Byte, or *Ignore. The latter has the advantage that it also allows a *NoIgnore* state to be set.

Related SWIs

OS_Byte 5 (page 2-20), OS_Byte 182 (page 2-23), OS_Byte 246 (page 2-29)

Related vectors

ByteV

OS_Byte 182 (SWI &06)

Reads/writes the printer *NoIgnore* state

On entry

R0 = 182 (reason code)
R1 = 0 to read or new state to write
R2 = 255 to read or 0 to write

On exit

R0 = preserved
R1 = state before being overwritten
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The state stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((state AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call allows reading the current *NoIgnore* state or changing it to a new value.

If the value read or written is &80 (ie has bit 7 set), then the printer ignore character is not used. If bit 7 is clear, then the current printer ignore character is filtered out.

The default setting of this flag is controlled by *Configure Ignore and may be changed temporarily using *Ignore.

Related SWIs

OS_Byte 6 (page 2-22), OS_Byte 246 (page 2-29)

Related vectors

ByteV

OS_Byte 199 (SWI &06)

Reads/writes the spool file handle

On entry

R0 = 199 (reason code)
R1 = 0 to read or new handle (as returned by OS_Find) to write
R2 = 255 to read or 0 to write

On exit

R0 = preserved
R1 = handle before being overwritten
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call reads/writes the spool file handle, which sets the destination of spooled data. The handle must have been correctly returned from a previous call to OS_Find (page 3-68). If the file handle is zero, or if spooling is disabled by OS_Byte 3, then no spooled data is sent.

Related SWIs

OS_Byte 3 (page 2-18), OS_Find (page 3-68)

Related vectors

ByteV

OS_Byte 236 (SWI &06)

Read/write character destination status

On entry

R0 = 236 (reason code)
R1 = 0 when reading or new status when writing
R2 = 255 to read or 0 to write

On exit

R0 = preserved
R1 = status before being overwritten
R2 = cursor key status (see OS_Byte 237, page 2-405)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The status stored is changed by being masked with R2 and then exclusive ORd with R1. ie ((status AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call reads and writes the output streams value. This can also be written by OS_Byte 3. See OS_Byte 3 for a list of the bit values.

Related SWIs

OS_Byte 3 (page 2-18)

Related vectors

ByteV

OS_Byte 245 (SWI &06)

Read printer driver type

On entry

R0 = 245 (reason code)
R1 = 0
R2 = 255

On exit

R0 = preserved
R1 = value before being overwritten
R2 = value printer ignore character (see OS_Byte 246, page 2-29)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored must not be changed by making R1 and R2 other than the values stated above. Use OS_Byte 5 instead to write.

This call will return values in R1 in the following range:

Value	Type
0	Printer sink
1	Parallel (Centronics) printer driver
2	Serial output
3 - 255	Files in system variables PrinterType$*n* (eg the NetPrint module uses PrinterType$4)

This call does not wait for the printer buffer to empty first. Because of this, it does not enable interrupts, and so may be used to read the printer type from within an interrupt routine.

Related SWIs

OS_Byte 5 (page 2-20)

Related vectors

ByteV

OS_Byte 246 (SWI &06)

Read/write printer ignore character

On entry

R0 = 246 (reason code)
R1 = 0 to read or new ASCII value to write
R2 = 255 to read or 0 to write

On exit

R0 = preserved
R1 = value before being overwritten
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call allows reading the current state of the printer ignore character or changing it to a new value.

Related SWIs

OS_Byte 6 (page 2-22), OS_Byte 182 (page 2-23)

Related vectors

ByteV

OS_PrettyPrint (SWI &44)

Write an indirect string with some formatting to all of the active output streams

On entry

R0 = pointer to null-terminated string to write
R1 = pointer to dictionary (0 means use the internal RISC OS dictionary)
R2 = pointer to null-terminated special string

On exit

R0 = preserved
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call acts like OS_Write0 (page 2-16), with several differences:

- Several characters have special meanings to OS_PrettyPrint (page 2-30).
- It will break a line at a SPACE (ACSII 32) if the next word will not fit on the line; it will not do this at hard spaces.
- Compacted text is handled.

The following characters in the string have special meanings:

- CR (ASCII 13) causes a newline to be generated.
- TAB (ASCII 9) causes a tabulation to the next multiple of eight columns.
- SPACE (ASCII 31) is a hard space.

- ESC (ASCII 27) indicates that a dictionary entry should be substituted.

Compacted text uses an escape character in the print string to indicate a dictionary entry. It is followed immediately by a byte which is the dictionary entry number. If this byte is in the range 1 to 255, then the appropriate string in the dictionary is substituted. If it is 0, then the special string pointed to by R2 on entry is substituted. (This is used in particular by the *Help command.)

The format of a dictionary is a linear list of entries, which can recursively refer to other dictionary entries; each entry is a length byte followed by a null-terminated string. This means that a dictionary does not have to have 255 entries. It can be ended at any point with a zero length entry.

The contents of the RISC OS dictionary is summarised below:

Token	String
0	_string pointed to by R2_
1	"Syntax: *"_string pointed to by R2_
2	" the "
3	"director"
4	"filing system"
5	"current"
6	" to a variable. Other types of value can be assigned with *"
7	"file"
8	"default "
9	"tion"
10	"*Configure "
11	"name"
12	" server"
13	"number"
14	"Syntax: *"_string pointed to by R2_" <"
15	" one or more files that match the given wildcard"
16	" and "
17	"relocatable module"
18	_CR_"C(onfirm)"_TAB_"Prompt for confirmation of each "
19	"sets the "
20	"Syntax: *"_string pointed to by R2_" [<disc spec.>]"
21	"]"_CR_"V(erbose)"_TAB_"Print information on each file "
23	"spriteLandscape [<XScale> [<YScale> [<Margin> [<Threshold>]]]]"
24	" is used to print a hard copy of the screen on EPSON-"
25	"."_CR_"Options: (use - to force off, eg. -"
26	"printe"
27	"Syntax: *"_string pointed to by R2_" <filename>"
28	"select"
29	"xpression"
30	"Syntax: *"_string pointed to by R2_" ["
31	"sprite"
32	" displays"
33	"free space"
34	" (off)"
35	"library"
36	"parameter"
37	"object"

```
38      " all "
39      "disc"
40      " to "
41      " is "
```

Related SWIs

OS_WriteC (page 2-13)

Related vectors

None

OS_PrintChar (SWI &5D)

Send a character to the printer stream

On entry

R0 = character to print

On exit

R0 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call will send a character to the printer. OS_Bytes 3 and 5 control whether there is a printer selected and which device it is.

Note that the printer ignore character (see OS_Byte 6, page 2-22) is not used by this call.

Related SWIs

None

Related vectors

None

OS_WriteN (SWI &46)

Write a counted string to the VDU

On entry

R0 = pointer to string to write
R1 = number of bytes to write

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

If the VDU is the only active stream, this call uses the low-level VDU drivers directly, and is therefore much more efficient than using multiple calls to OS_WriteC. Also, because no special character is used to mark the end of the string, any VDU sequence may be sent.

Related SWIs

None

Related vectors

WrchV

OS_WriteI (SWI &100–1FF)

Write an immediate byte to all of the active output streams

On entry

—

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call writes the character contained in the bottom byte of the SWI number, using OS_WriteC. It has the advantage of being more compact and quicker for a program using it than the equivalent usage of OS_WriteC. For example, to write a 'J' character, you would use:

```
SWI     OS_WriteI + ASC"J"
```

Related SWIs

OS_WriteC (page 2-13)

Related vectors

WrchV

*Commands

*Configure Ignore

Sets the configured printer ignore character

Syntax

```
*Configure Ignore [ASCII_code]
```

Parameters

ASCII_code	ASCII code, from 0 to 255

Use

*Configure Ignore sets the configured printer ignore character to the specified ASCII code. This character is filtered out when printing is enabled via the VDU printer stream or OS_Byte 5.

The default value is 10 (ASCII linefeed). On some printers, you may find this causes lines to overprint each other, in which case you should omit the ASCII code so all characters are sent to the printer. *Configure Ignore 0 will not ensure all characters are printed; it will set the configured printer ignore character to ASCII 0 (the null character).

The change takes effect on the next hard reset.

Example

`*Configure Ignore 10`	*Do not print ASCII character 10*
`*Configure Ignore`	*Print all characters*

Related commands

*Ignore

Related SWIs

OS_Byte 6 (page 2-22), OS_Byte 5 (page 2-20), OS_Byte 182 (page 2-23), OS_Byte 246 (page 2-29)

Related vectors

None

*Configure Print

Sets the configured default destination for printed output.

Syntax

```
*Configure Print n
```

Parameters

n	0 to 7

Use

*Configure Print sets the configured default destination for printed output. The values of n correspond to the following printers:

n	Printer
0	Printer sink (no output)
1	Parallel port
2	Serial port
3	User printer driver
4	Network printer (handled through NetPrint)
5-7	Files in system variables PrinterType$5, 6 or 7

The change takes effect on the next hard reset.

Example

`*Configure Print 1`	*select the parallel printer port*

Related commands

None

Related SWIs

OS_Byte 5 (page 2-20)

Related vectors

None

*Ignore

Sets the printer ignore character.

Syntax

```
*Ignore [ASCII_code]
```

Parameters

ASCII_code	ASCII code, from 0 to 255

Use

*Ignore sets the printer ignore character to the specified ASCII code. This character is filtered out when printing is enabled via the VDU printer stream or OS_Byte 5.

The default value is 10 (ASCII linefeed). On some printers, you may find this causes lines to overprint each other, in which case you should omit the ASCII code so all characters are sent to the printer. *Ignore 0 will not ensure all characters are printed; it will set the printer ignore character to ASCII 0 (the null character).

OS_Byte 6 performs the same action as this command; OS_Byte 246 also reads and writes the printer ignore character. OS_Byte 182 controls the *NoIgnore* state.

Example

`*Ignore 10`	*Do not print ASCII character 10*
`*Ignore`	*Print all characters*

Related commands

None

Related SWIs

OS_Byte 5 (page 2-20), OS_Byte 6 (page 2-22), OS_Byte 182 (page 2-23), OS_Byte 246 (page 2-29)

Related vectors

None

21 VDU Drivers

Introduction

Though strictly speaking part of the character output system, the VDU drivers are quite complex, and deserve a chapter of their own. This chapter introduces the important concepts relating to the VDU, such as:

- screen modes
- graphics and text windows
- colour palette
- colour patterns
- the mouse
- putting text and graphics on the screen
- multiple display pages

The chapter entitled *Character Output* on page 2-1 described how to write to the VDU. This chapter describes what special effects occur when particular characters are sent.

There are also a large number of VDU specific commands that allow fine control of its operation.

There are five important aspects of VDU interaction which are not described in this chapter. These are:

- the Font manager
- the Window manager
- the Draw module
- Sprites
- the ColourTrans module

These are implemented as modules separate from the RISC OS kernel, and are described in their own chapters.

Overview

The most important call relating to the VDU is OS_WriteC, as this is used in nearly all programs which have to output to the screen. Other calls can be used for more direct control of the VDU facilities.

The VDU display on RISC OS comes from the VIDC chip. This reads the contents of a block of memory and converts it into a form that can drive a video monitor.

VDU commands

This chapter differs from others in this manual in that, in addition to a list of SWIs and *commands, there is also a list of VDU commands. To issue VDU commands, simply use OS_WriteC to send characters to the VDU stream. All characters are strictly VDU commands, but those between 0 and 31, and 127 are of special interest because they cause special actions to take place. The others are simply printed on the screen as a character.

These special characters are used as commands. They can be followed by a sequence of characters, the length of which depends on the command. In some cases, the character on its own is sufficient, but it can require up to 9 following bytes to complete the command. These bytes are queued until the required number are in the queue before the command is executed.

To represent these sequences of characters sent to the VDU using OS_WriteC, a shorthand is used in this chapter. You will see VDU followed by numbers separated by commas. This represents each character being sent through OS_WriteC.

For example, VDU 65 sends character 65, an ASCII 'A', to OS_WriteC. VDU 17,3 sends character 17 followed by character 3.

Modes

RISC OS supports many different ways of displaying information on the screen. Each of these different ways is called a mode. The exact number of modes available depends on the type of monitor you have. They are all bit-mapped displays, in which one or more bits of screen memory control the colour of a dot, or pixel, on the screen. Two main characteristics distinguish the modes.

- The resolution of a mode relates to the number of pixels which can be displayed horizontally and vertically.
- The number of colours that can be displayed at once is determined by the number of bits used to store each pixel. Typically, this can be 1, 2, 4 or 8 bits, leading to 2, 4, 16 or 256 colours on the screen at once.

Between them, the resolution and number of colours determine the amount of screen memory used by a mode.

A complete list of the available modes is given in the description of VDU 22 on page 2-85, which is the command that changes modes.

Text and graphics

There are two distinct types of object that the VDU drivers can draw onto the screen.

- The text VDU deals with drawing text characters
- The graphics VDU handles any arbitrary drawing of dots, lines, shapes, etc.

Windows

Different commands will act to either text or graphics areas. Each has a window, or area where their output will go. After a mode change, both text and graphic windows fill the screen and overlap each other exactly. There is no conflict in having them overlap, since the window is just a declaration of boundaries. Either window can be changed at any time to be any size. Any output to a window will be clipped to it. For example, if only part of a line appears in the graphics window, then only that part will be shown and the rest ignored.

A cursor is the place at which the next output will go. There are independent text and graphics cursors, which must remain inside their relevant window.

Various control commands are provided to affect the output in text and graphics windows. Examples of such actions are:

- changing the colours in which output occurs,
- moving the appropriate cursor,
- clearing the window.

Text VDU

Text characters are patterns of pixels which are positioned on the screen at character-aligned positions. That is, the screen is treated like an array of character sized boxes, into which can go any printable character.

All text display is normally confined to the text window. All scrolling is confined to this region, sometimes called the scrolling window, because text can be scrolled within it. The graphics window cannot be scrolled automatically; but you can use block move to perform scrolling.

The text cursor shows the position on the screen of the next character to be displayed. This is usually a flashing underline. There can be a second cursor which is used with cursor editing (this is described later).

Note that there are some screen modes that will only display text.

Graphics VDU

The graphics VDU handles the drawing of objects such as points, lines, circles, ellipses, etc. The graphics window, like the text window, starts as the whole screen after a mode change. The graphics cursor, which is invisible, marks the last point at which a graphics operation ended.

Joining text and graphics

The VDU driver can be configured to print text at the graphics cursor instead of the text cursor. This means that text will be drawn using the current graphics cursor for positioning, and using the graphics colour, etc. The advantage of this mode is that it enables characters to be drawn at any pixel alignment, and to be clipped to the graphics window (important when you use the Wimp environment). The disadvantages are that the characters take longer to draw and scrolling is not available. Generally, when text is printed at the graphics cursor, this is referred to as VDU 5 mode because this is the command that enables it.

Cursor editing

Although the cursor editing facility isn't strictly part of the VDU drivers, its presence does have some interaction with the VDU.

Usually there is only one text cursor, but when you press one of the four cursor direction keys, cursor editing mode starts. There are now two cursors; the output cursor, which is now shown as a steady 'blob', and the input cursor, which is an underline flashing at twice the usual rate. The Copy key has the action of copying what is under the input cursor to the output cursor as if it was typed.

See the chapter entitled *Character Input* on page 2-337 for a full description of these keys and their control.

Cursor editing mode is not available in VDU 5 mode, and it is cancelled when you send an ASCII 13 (carriage return) to the VDU stream. This is usually done when you press Return at the end of an input line.

Colours

The number of colours available on the screen at any time is either 2, 4, 16 or 256. When you first enter a mode, the default colours are assigned. These can subsequently be changed with the palette.

256-colour modes

In 256 colour modes, there are 64 different colours, and each colour may have four different shades, resulting in a total of 256 different colours.

Foreground and background

You may choose to display your text or graphics in a different colour from the defaults. To do this, there are commands to change the foreground and background of each. Usually, the foreground colour is that in which the text or graphics drawing is done, and the background colour is used for all other drawing, such as a screen clear. RISC OS can be changed so that the background colour is used for drawing if required.

The palette

Another important part of the VDU is the palette. This is the control of what colours appear on the screen. The palette is a table built into the VIDC chip which determines the relationship between the colour number stored in the screen memory (logical colour), and the actual colour information sent to the monitor (physical colour). Care should be taken not to confuse logical and physical colours. Thus, while colour 0 on RISC OS is black by default, it can be made to be any colour by changing how the palette maps it.

The palette is programmed in terms of the intensity of the signal on each of the red, green and blue guns in a colour monitor. These intensities have 4 bits each, which gives twelve bits altogether, hence the 4096 (2^{12}) physical colours. Flashing colours are accomplished by a logical colour having 2 physical colours associated with it. These are swapped at a programmable rate, causing flashing.

The palette also controls the colour of the border around the screen and the colours of the mouse pointer. These can be set independently of any other colour on the screen. The border and mouse colours are always 2bpp (4 colours) in all screen modes.

Tints

In 256 colour modes, each pixel is represented by an 8-bit value. Six bits are the logical colour, and the other two bits are the tint. The tint is a direct control of the amount of grey which is added to the base colour, to one of 4 levels.

The six bits in the logical colour set the basic colour from the range of different shades of colours provided by the palette. The tint is the fine control within this range.

ECF patterns

The Extended Colour Fill patterns are a means of increasing the apparent number of colours by producing a fine chequerboard mix of colours. This is of most use in modes where there are few colours available, because it gives the effect of having more colours on the screen than there are.

Four different ECF patterns are provided, and can be independently defined.

Normally, the origin of the ECF patterns is based on the bottom left corner of the screen. This can be changed, so that it aligns with any point on the screen, such as the current graphics window.

Bell

The VDU drivers control how the bell will sound. The bell is a sound that is made when the standard ASCII character 7 (Ctrl-G) is sent to the VDU. Its volume, pitch and duration can all be customised.

Mouse and pointer

The mouse is a device that is moved on a surface, rolling an internal ball, usually with several buttons. The pointer is a reflection on the screen of the mouse's movements. Normally, it appears as a small arrow, but can be programmed to be any shape. It is also possible to disconnect the pointer from the mouse and move the pointer to where the program wants it to be. This is useful when switching between windows under program control.

RISC OS provides control over how much the pointer moves in response to a mouse movement. This sensitivity control can be useful in situations where fine or coarse movement is required by different programs.

Screen configuration

Full control is given over how video information is generated. Depending on how it looks on screen, the display can be shifted up or down. Some monitors do not allow for this adjustment, so this facility is provided.

Also, the interlace can be switched on or off. Interlace means that images sent to a monitor alternate one scan line up and down on alternate frames. On a monitor which has a long persistence phosphor (images take some time to fade), an interlaced image eliminates the 'lined' effect of a screen image. On a short persistence screen interlace can cause a flicker, because the first image has faded before the second one is finished.

RISC OS supports many different kinds of monitor. Depending on the type of monitor used, only a subset of all possible modes are available on it. Thus there is a command to set which monitor is connected, so that incorrect modes are not accidentally entered.

Multiple banks

Normally, there is one bank of memory that is used for the screen. If it is changed, then this is reflected on the screen as it is refreshed by VIDC. Sometimes it is useful to write to one bank of screen memory, while another is displayed and then swap when finished. This produces an 'instant draw' effect, which is visually pleasing.

Whilst normally only two banks would be used for this kind of application, you can have as many banks as will fit in the allocated screen RAM area. This requires copies of the screen RAM requirements for each bank. For example, with two banks of screen memory, an 80K mode will require 160K.

Writing to the screen

Many different kinds of things can generate output on the screen, or more strictly speaking, the current screen bank. Text or graphics can be written, and many commands exist to alter where and how output will appear on the screen.

Writing text

Sending printable characters through OS_WriteC (page 2-13) will result in it appearing at the text cursor position in the current window. It will wrap around to following lines when it reaches the right hand side of the window. Certain control commands can move the text cursor in all directions or to a given place in the window. Usually, the cursor moves right after a character is printed. This can be changed so it moves in any of the four directions.

Writing graphics

Many different kinds of graphics can be put onto the screen, such as:

- circles, ellipses, arcs, segments, and sectors
- triangles, rectangles and parallelograms
- filled areas, such as all those above and any irregular shape
- dots
- solid and dotted lines
- text in VDU 5 mode

See the chapter on sprites to see how any sized array of pixels can be written to the screen.

As well as different shapes, there is control over how it is written over what is already on the screen. It can be configured to:

- overwrite existing graphics,
- OR with it,
- AND with it,
- exclusive OR with it,
- invert it,

and so on.

As well as having control over the colour and writing mode, you can use any of the ECF patterns to write with.

Clearing the screen

The graphics or text windows can either be completely or partially cleared. This will be done with the current graphics or text background colour as appropriate.

Synchronised writing

There is a mechanism under RISC OS of waiting until a Vsync event occurs and then writing to the screen. This can make screen update very smooth, as writing to the screen memory does not clash with the VIDC chip reading it to send to the monitor. If they do clash, then a 'tearing' can appear briefly. This is because one part of the memory being written to is displayed in its old state and the other part in the new.

Unless you plan to use multiple paging techniques, then this is a good way of achieving smooth animation.

Reading from the screen

As well as writing to the screen, it is possible to read some information back from it. There is a command to read a character from under the text cursor and work out what its ASCII value is. Cursor editing uses this facility.

You can also read the logical colour and tint of a point. Given that there is another call to return the palette setting for a colour, it is easy to combine the two and work out the 'real' colour of pixels on the screen.

The screen can be saved as a file, which can be subsequently treated as a sprite, or edited with Paint for example. There is a corollary command to load it back onto the screen.

Information about the VDU

There are a number of calls to get all kinds of information about the configuration and status of the VDU driver. Here is some of the information that can be read:

- size and position of graphics and text windows
- position of graphics and text cursors
- description of current screen mode
- size of screen memory
- palette mapping
- foreground and background text and graphics colours
- banks used by VDU and screen
- number of bytes queued for a VDU command being composed
- number of lines printed since last page halt
- in VDU 5 mode or not.

VDU extension vector

The normal VDU driver can be completely replaced with a custom driver if required. The VDU extension vector, called VDUXV, can be called instead of the normal VDU vector. This can be useful if you want to change the characteristics of screen output in a dramatic way. For example, the font manager module uses this to quickly display complex fonts. Going through the normal VDU mechanism would be too slow, because it would have to be done a dot at a time.

Technical Details

VDU commands

As mentioned earlier, 'VDU' followed by a series of numbers separated by commas is used in this chapter to represent a character being sent to OS_WriteC. For convenience, we will use the shortcuts that BBC BASIC uses with its VDU statement. Here is a brief reminder of the syntax of that statement:

VDU n sends character n to OS_WriteC. VDU m,n sends ASCII m followed by ASCII n.

VDU n; sends the number n as two bytes, first n MOD &100, then n DIV &100. This sends 16-bit numbers to the VDU drivers; eg coordinates in graphics commands.

VDU n| sends n as a single byte, followed by nine 0 bytes. This is used as shorthand in calls in which not all of the parameter bytes are needed. As nine is the largest number of bytes required by any VDU sequence, ending the command with '|' guarantees enough bytes to complete it. Any extra zeros are ignored by the VDU drivers.

Of course, as long as the correct characters are sent to the VDU, it doesn't matter how they get there. For example, an assembly language equivalent to VDU 12 (clear screen) is:

```
SWI OS_WriteI+12
```

The effect is the same in both cases.

Screen modes

When changing mode, a great many things are initialised. For a complete list of these and other mode notes, see VDU 22.

*Configure Mode will set up the screen mode to be used after a hard reset.

When a program wishes to change mode, it must check that there is enough memory allocated for the screen for that mode and that the monitor being used is compatible with the mode. OS_CheckModeValid (page 2-223) must be called to check these two things. If you don't, then VDU 22 will do it anyway, but it is better for the program to be aware of what's happening. If the mode requested cannot be used, OS_CheckModeValid will also return a suggestion for a mode to use in place of it.

Screen configuration

The computer can adjust its output to suit its attached monitor in a number of ways.

*Configure MonitorType (page 2-232) is used to tell RISC OS what kind of monitor is attached, since the system has no way of detecting this from hardware. This command allows the system to subsequently disallow any modes that are not compatible with the attached monitor.

*Configure Sync (page 2-239) will set up the vertical sync output of the video connector to be vertical or composite sync. Different monitors may require either of these, though most use composite sync.

*Configure TV (page 2-240), *TV (page 2-242) and OS_Byte 144 (page 2-158) can all adjust the position of the video output up or down by several lines, and switch interlace on and off. VDU 23,0 can also control the interlace setting.

Multiple bank modes

There are two main commands that can be used to handle multiple banks of screen memory. OS_Byte 112 selects which bank of memory to send VDU output to. OS_Byte 113 selects which bank of memory is used by the VIDC hardware to write out to the screen. By using these two, it is simple to swap screens at will.

OS_Byte 250 reads the current OS_Byte 112 setting, and OS_Byte 251 reads the current OS_Byte 113 setting.

In order to use multiple banks, you will probably have to use *Configure ScreenSize to set the amount of memory to reserve for all the banks.

*Shadow exists mainly for compatibility with BBC/Master operating systems. Under RISC OS, it can select between two banks of memory to be used on the next mode change. OS_Byte 114 has the same effect as *Shadow. OS_Bytes 112 and 113 support the shadow system, but you are better off using bank numbers directly.

For those who want low level access to screen banks, OS_Word 22 allows setting the addresses of the VDU bank and the VIDC bank directly.

Colours

These are the colours as set up after a mode change:

Two-colour modes

0 = black
1 = white

Four-colour modes

0 = black
1 = red
2 = yellow
3 = white

16-colour modes

0 = black
1 = red
2 = green
3 = yellow
4 = blue
5 = magenta
6 = cyan
7 = white
8 = flashing black-white
9 = flashing red-cyan
10 = flashing green-magenta
11 = flashing yellow-blue
12 = flashing blue-yellow
13 = flashing magenta-green
14 = flashing cyan-red
15 = flashing white-black

256-colour modes

256 colour modes are treated differently from the others. Instead of using the standard 16 entry physical colour table, there are two systems which are used by different commands. The internal format is the less easy to use of the two. In it, the bits are structured as follows:

Bit	Meaning
0	Bit 0 of palette index
1	Bit 1 of palette index
2	Bit 2 of palette index
3	Bit 3 of palette index
4	Red bit 3 (high)
5	Green bit 2
6	Green bit 3 (high)
7	Blue bit 3 (high)

where the palette index (0 - 15) controls which VIDC palette entry is used, but with some bits of the palette entry then being overridden by the top 4 bits of the memory byte. With the default palette setting, this becomes:

Bit	Meaning
0	Tint bit 0 (red + green + blue bit 0)
1	Tint bit 1 (red + green + blue bit 1)
2	Red bit 2
3	Blue bit 2
4	Red bit 3 (high)
5	Green bit 2
6	Green bit 3 (high)
7	Blue bit 3 (high)

Each primary colour has 4 bits of intensity, but the two least significant bits (the tint bits) are shared between the three colours. Therefore, some intensities of a primary colour (for example, red) can only be obtained at the expense of adding in a certain amount of grey.

The second form for 256 colours, which is used by some commands is structured as follows:

Bit	Meaning
0	Red bit 2
1	Red bit 3 (high)
2	Green bit 2
3	Green bit 3 (high)
4	Blue bit 2
5	Blue bit 3 (high)
6	Tint bit 0 (red + green + blue bit 0)
7	Tint bit 1 (red + green + blue bit 1)

The tint is controlled separately in most commands; bits 6 and 7 are only used in the native ECF setting, which is not often used in 256 colour modes.

This format is converted into the internal format when stored, because that is what the VIDC hardware recognises.

To change colour

VDU 17 (page 2-76) can be used to change the text colour. VDU 23,17,5l will exchange the text foreground and background colours.

VDU 18 (page 2-77) can change the graphics colour, and much more than just that. Because graphics can interact with what is already is on the screen, then VDU 18 can set up the graphics to be ORd, ANDed, XORd, inverted and so on.

In 256 colour modes, VDU 23,17,0 - 3 can be used to set the tints to be used when next printing/plotting.

Palette

VDU 19 (page 2-79) can be used to change the way that the palette defines the logical to physical colour relationship. It has many modes and as well as changing the logical colours, can also set the border, flashing and cursor colours. OS_Word 12 can also be used to write the palette.

VDU 20 (page 2-83) will return the palette to the condition that it was just after a mode change. This would be used by a program just before finishing, if it had altered the palette during running.

If you want to read the palette setting of a colour, OS_ReadPalette or the BBC/Master compatible OS_Word 11 can be used.

Flashing colour

RISC OS will swap two colours at a programmed interval. If they are the same colour, then there is no noticeable effect. If they are different, then flashing will result. VDU 19 can individually set these colours to be any colour from the palette.

The speed at which flashing occurs can be controlled by OS_Bytes 9 and 10. They set the duration in video frames. VDU 23,9 and VDU 23,10 have the same effect as these calls. The duration settings can be read by OS_Bytes 194 and 195.

OS_Byte 193 allows a program to read or alter the flash counter. This is a decrementing counter that swaps colours when the count reaches zero.

ECF patterns

There are several different ways of changing ECF patterns. The main command is VDU 23,2-5. This can operate in two modes depending on the setting of VDU 23,17,4. Also, VDU 23,12-15 can be used for simpler patterns.

Colours and resolution

Both commands are passed 8 bytes that define the pattern. The number of pixels depends how many colours are available in the screen mode you are using:

Colours	**Number of pixels set by each line**	
available	**VDU 23,2-5**	**VDU 23,12-15**
2	8	2
4	4	2
16	2	2
256	1	1

You can see that while the number of pixels in the pattern diminishes, the number of potential colours increases.

256 colour patterns

As you can see, in a 256 colour mode, the pattern is simply a colour description for each line. VDU 23,2-5 uses the internal 256 colour map, while VDU 23,12-15 uses the simpler colour map. When stored, the internal form is used. This should be borne in mind if you use OS_Word 10 to read the ECF definitions.

VDU 23,12-15

This call uses a simpler pattern. The 8 parameters passed form a pattern as follows:

1	2
3	4
5	6
7	8

So it describes a simple 2 by 4 pattern for all but 256 colour modes. Here it is one colour per line, for all 8 lines, like VDU 23,2-5.

VDU 23,2-5

This call is more complex. It uses one line per parameter, and there is a direct trade-off between colours and resolution. Thus, for a 2 colour mode, it can display an 8 by 8 pattern of on or off pixels, in 256 colour mode, it can only generate 8 lines of a single different colour each.

VDU 23,17,4

VDU 23,17,4 is used to select between BBC/Master compatible mode and native RISC OS mode. These modes describe how ECF colour descriptions are mixed when using VDU 23,2-5. For some examples, see the section entitled *Application Notes* on page 2-243.

Initialisation

VDU 23,11 will reset the ECF pattern definitions to their default values. It will also reset the VDU 23,17,4 flag to the default BBC/Master compatible state.

Setting the origin

By default, patterns are written as if their bottom left hand corner aligned with the bottom left hand corner of the screen. Using OS_SetECFOrigin, you can adjust this to be any point on the screen. Thus, an ECF pattern can now be aligned with any object, such as the graphics window. VDU 23,17,6 has the same effect as this call.

Bell

The bell can be made to sound by sending a VDU 7 to OS_WriteC.

To configure how it will sound:

- OS_Byte 211 (page 2-169) will select the sound channel used
- OS_Byte 212 (page 2-170) will adjust the volume
- OS_Byte 213 (page 2-172) will adjust the frequency
- OS_Byte 214 (page 2-174) will adjust the duration

*Configure Quiet will select a medium volume, while *Configure Loud will select the loudest volume.

Cursors

VDU 5 will link text and graphics cursors and cause all subsequent output to be printed at the graphics cursor position. This command can be cancelled using VDU 4. The text input cursor is normally displayed unless disabled by VDU 23,1. Both this and VDU 23,0 can be used to change the appearance of the cursor.

There are a number of VDU commands that affect the position of the text cursor directly:

- VDU 30 (page 2-125) – send the text cursor to its home position, which is usually the top left corner of the current window.
- VDU 31 (page 2-126) – set the text cursor to any position on the screen.
- VDU 8 (page 2-67) – back space
- VDU 9 (page 2-68) – horizontal tab
- VDU 10 (page 2-69) – line feed. That is, move down.
- VDU 11 (page 2-70) – vertical tab. That is, move up one line.
- VDU 13 (page 2-72) – move back to the start of the line.

- VDU 127 (page 2-127) – delete. That is, backspace, print a space then backspace again

The position of the text cursor can be read with OS_Byte 134. If cursor editing is in progress, then OS_Byte 165 can be used to read the position of the output cursor, usually displayed as a solid blob.

Normally, when a character is printed, the cursor currently used will move to the right. This action can be controlled by VDU 23,16. It can set the cursor to move in any of four directions. It also controls how cursors act at the end of lines, and so on.

OS_RemoveCursors will remove the input and output cursors and store their state internally. A subsequent call to OS_RestoreCursors will restore them exactly. These calls are used mainly by low-level draw routines to avoid mixing the cursors with what is drawn on the screen.

OS_Word 13 will return the current and previous graphics cursor positions. Using OS_ReadVduVariables, even earlier coordinates can be read.

Mouse and pointer

When a mouse button is pressed or released a record is kept in the mouse buffer. OS_Mouse will read a mouse record from this buffer. It stores the position of the mouse, the state of its buttons and the time the record was put into the buffer. OS_Byte 128 can also be used for this as well as reading how much free space is in the mouse buffer.

OS_Word 21,3 will set the mouse position, so subsequent writes to the mouse buffer will assume the mouse is at the specified location, and move from there.

OS_Word 21,4 will read the unbuffered mouse position. That is, where it is at the moment of calling this function. This bypasses the buffer, so subsequent reads of the buffer may not tie up with this position. It is better to use one or the other method exclusively in a program.

Pointer

The ratio of mouse movement to pointer movement on screen can be controlled by OS_Word 21,2 or permanently set by *Configure MouseStep.

The pointer that appears on the screen can be defined in four shapes. OS_Word 21,0 can define the shape and colour of each of these. OS_Byte 106 is used to select which pointer to use, or switch it off completely. *Pointer can also be used to switch it on or off.

The pointer will be confined to the box defined by OS_Word 21,1. This would usually be set to the graphics window.

The pointer's position on the screen can be set with OS_Word 21,5 and read with OS_Word 21,6.

Getting Information

There are many ways of extracting information about the state and configuration of the VDU system.

OS_Byte 217 will read the number of lines since the display was last stopped scrolling if it was in paged mode.

OS_Byte 218 returns how many bytes are in the VDU queue. This is used when a multiple byte VDU command is being collected.

OS_Byte 163 will return the current dot-dash line length and the amount of memory allocated for sprites. It can also set the dot-dash length.

OS_ReadDynamicArea is a better way to read the amount of memory allocated for system sprites – this call will also return the memory allocated for screen bank use.

OS_Byte 117 reads the VDU status. This involves:

- whether the printer output is enabled
- if paged scrolling is enabled
- if in shadow mode
- if in VDU 5 mode
- if cursor editing
- if the screen is disabled with VDU 21

OS_ReadVduVariables provides a large number of variables that can be read. OS_Byte 160 is a subset of this, kept for BBC/Master compatibility reasons. Almost all information about windows, cursors and colours can be accessed here. Two special variables provided are a pointer to a fast horizontal line draw routine and access to colour blocks.

OS_ReadModeVariable returns the fixed information about a mode, such as how many pixels across and down it is, and how many colours it supports.

Reading from the screen

OS_Byte 135 will read the ASCII value of the character at the text cursor position and also reads the current screen mode.

OS_ReadPoint will read the logical colour of a pixel. OS_Word 9 performs much the same function, but is kept mainly for compatibility with BBC/Master series.

*ScreenSave will copy the screen contents into a file where it can subsequently be edited with Paint or reloaded to the screen with *ScreenLoad.

Writing to the screen

Output to the screen can be disabled by VDU 21. It can be restored by VDU 6.

VDU 26 will restore the graphics and text windows to their default states. That is, both filling the screen.

Text

Text can be sent to the screen with any VDU command from 32 to 255, excepting 127 which is the delete command.

VDU 28 defines the text window. VDU 12 will clear the window that the text cursor is in. After a VDU 12, the text cursor is moved to its home position, usually the top left hand corner. VDU 23,8 will clear a block within the text window.

Page mode means that when about 75% of a screenful has been shown, then the system will pause and wait for Shift to be pressed before starting again. This stops text being lost from scrolling off the top of the screen too quickly Paged mode can be enabled by VDU 14 and disabled with VDU 15. By default, paged mode is off.

*Configure Scroll and NoScroll configure whether text will scroll when it reaches the bottom of the text window. This means that when NoScroll is set a character can be printed at the bottom right of the screen without immediately scrolling the screen. This feature can also be controlled with VDU 23,16 and allows a full screen of text to be simply printed.

VDU 23,7 can scroll the text window or the whole screen in any direction.

In VDU 5 mode, it is possible to change the size and spacing of text with VDU 23,17,7. This is how you would generate a message with large gaps between the characters.

Redefining characters

Each printable character (one that is not a command) is an array of 8 by 8 pixels that is defined in the shape of standard ASCII and ISO characters. All of these characters can be redefined to be any pattern.

To change the definition of a printable character, VDU 23,32-255 must be used. The character number that you wish to redefine is the second parameter, in the range 32-255. It is followed by 8 bytes that define the bit pattern to be used.

OS_Byte 20 will reset all character definitions to their default. OS_Byte 25 will reset a given group of them. OS_Word 10 can read the definition of any character from the current system font.

Printer

VDU 1 will send the following character to the printer stream. VDU 2 will enable the stream, so that all characters sent to the VDU are also sent to the printer stream. This state can be disabled by VDU 3.

Graphics

VDU 24 will define the position of the graphics window. VDU 16 will clear it to the current graphics background colour.

VDU 25 is the main graphics plot command. OS_Plot has the same effect as it, but is much faster, avoiding the delays inherent in the VDU stream. They both have a type parameter followed by x and y coordinates. The type covers moving the graphics cursor, plotting points, lines (solid and dotted), triangles, rectangles, parallelograms, circles, arcs, sectors, segments, ellipses and other graphic forms. These figures can be hollow or filled with the graphics foreground colour. It handles relative or absolute drawing That is, the x and y are relative to the current x and y or moving to a new absolute position on the screen.

When plotting dotted lines, the default pattern is a dot-space pattern repeated. This can be changed to any pattern. VDU 23,6 is passed 8 bytes that define a pattern up to 64 bits in length to be repeated. OS_Byte 163 sets how many bits are to be used. Simple patterns like &FF (solid line), &AA (the default dot-space) and &EE (dashed line: dot-dot-dot-space) can be used or any more complex pattern up to 64 bits in length. OS_Word 10 can read the current definition.

VDU 29 sets the graphics origin. This is the point on the screen that becomes the 0,0 point for all subsequent graphics operations.

OS_ChangedBox will tell you what area of the screen has been changed. This can be used to reduce the amount of redrawing that needs to be done by an application.

*ScreenLoad complements *ScreenSave, discussed earlier and load a file into the screen memory.

Vsync

OS_Byte 19 will wait until a Vsync occurs before returning. This allows programs that are quick enough to write to the screen without any kind of flickering or tearing of images.

VDU Calls

VDU 0

Null Operation

Syntax

VDU 0

Parameters

—

Use

VDU 0 does nothing. It is this that enables the '|' character in the VDU statement to work. Any of the nine zeros that are sent which aren't required by the current VDU command are 'swallowed up'.

VDU 1

Next character to printer only

Syntax

`VDU 1,`*`character`*

Parameters

character	to send to the printer stream

Use

VDU 1 sends the next character to the printer stream only, provided that the printer has been enabled by VDU 2. Otherwise, the next character is ignored. This enables the printer ignore character, and any other character which is not usually passed on by the VDU printer driver, to be sent to the printer through the VDU.

Example

`VDU 1,10`	*Send a line feed to the printer stream, if enabled*

VDU 2

Enable printer stream

Syntax

`VDU 2`

Parameters

—

Use

VDU 2 enables the printer stream. After this call, most characters sent to the screen will also be sent to the currently selected printer device. OS_Byte 5 controls this, and is described in the character output chapter. Only characters in the following ranges are sent to the printer: 32 - 126, 128 - 255 (ie the printable characters), 8 - 13 (backspace, horizontal tab, line feed, vertical tab, form feed and carriage return, respectively). No multi-byte control sequences, except the argument of VDU 1, are sent to the printer.

Even if the VDU drivers are disabled (using VDU 21) the characters sent to the VDU drivers will still be sent to the printer although they will no longer affect the screen. However, if the VDU is disabled using OS_Byte 3, then VDU 2 printing will not take place.

The effect of VDU 2 can be cancelled using VDU 3.

You can determine whether VDU printing is enabled using OS_Byte 117.

VDU 3

Disable printer stream

Syntax

```
VDU 3
```

Parameters

—

Use

VDU 3 cancels the effects of VDU 2 so that all subsequent printable characters are not passed through the kernel printer driver.

VDU 4

Split cursors

Syntax

```
VDU 4
```

Parameters

—

Use

VDU 4 cancels VDU 5 mode. It causes all subsequent printable characters to be printed at the current text cursor position using the current text foreground and background colours. The text cursor is normally displayed (unless it has been disabled using VDU 23,1) and after each character has been printed the cursor moves on by one character. The direction of cursor movement is normally to the right but may be altered using VDU 23,16.

After a character has been printed at the end of a row (or column if vertical printing is used) the cursor moves on to the start of the next screen line (or column), scrolling the screen when there are no more rows (or columns), providing scrolling is enabled. Cursor editing is allowed in this mode.

You can determine whether the cursors are split or joined using OS_Byte 117 (page 2-153).

VDU 5

Join cursors

Syntax

```
VDU 5
```

Parameters

—

Use

This enters VDU 5 mode. It links the text and graphics cursors and causes all subsequent printable characters to be printed at the current graphics cursor position, the topmost row, lefthand edge of the character being placed there. Characters are displayed in the current graphics foreground colour using the current graphics action. The background pixels in the character shape are not plotted.

You can set the character sizing and spacing using VDU 23,17,7...

After the character has been printed, the graphics cursor is moved by one character position. The direction of cursor movement is normally to the right but may be altered (using VDU 23,16). It moves to a new row (or column if vertical printing is being used) when necessary, or to the opposite corner of the graphics window if there are no more rows (or columns). Scrolling does not occur.

This command allows characters to be placed at any position on the screen, but means that the text is printed somewhat slower than when the cursors are split. In addition, each character is superimposed onto the existing text or graphics. Hence, printing a backspace character followed by a space moves the graphics cursor back by one character and then superimposes a space onto the character already there, thereby leaving it unaltered.

Cursor editing is not possible in this mode.

VDU 5 has no effect in text-only or Teletext modes. In other modes it may be cancelled using VDU 4.

VDU 6

Enable screen output

Syntax

```
VDU 6
```

Parameters

—

Use

VDU 6 restores the functions of the VDU driver after it has been disabled by VDU 21. It causes all subsequent printable characters to be sent to the screen and control sequences to be obeyed.

You can determine whether the VDU is enabled or disabled using OS_Byte 117 (page 2-153).

VDU 7

Bell

Syntax

```
VDU 7
```

Parameters

—

Use

VDU 7 generates either the default bell sound (as specified by *Configure Loud/Quiet and *Configure SoundDefault) or the bell sound defined using OS_Bytes 211 - 214.

VDU 8

Back space

Syntax

```
VDU 8
```

Parameters

—

Use

VDU 8 causes either the text cursor (by default) or the graphics cursor (in VDU 5 mode) to be moved back one character position (ie in the negative X direction). This normally means moving it to the left but will be different if the direction of cursor movement is altered (using VDU 23,16).

If the cursor was at the start of a row (or column if vertical printing is used) then it is moved back to the end of the previous row (or column), scrolling the screen if necessary. It does not cause the last character to be deleted.

VDU 9

Horizontal tab

Syntax

```
VDU 9
```

Parameters

—

Use

VDU 9 causes either the text cursor (by default) or the graphics cursor (in VDU 5 mode) to be moved on one character position (ie in the positive X direction). This normally means moving it to the right but is different if the direction of cursor movement is altered (using VDU 23,16).

If the cursor was at the end of a row (or column if vertical printing is used) then it is moved on to the start of the next row (or column), scrolling the screen if necessary.

VDU 10

Line feed

Syntax

```
VDU 10
```

Parameters

—

Use

VDU 10 causes either the text cursor (by default) or the graphics cursor (in VDU 5 mode) to be moved on one line (ie in the positive Y direction). This normally means moving it down but is different if the direction of cursor movement has been altered (using VDU 23,16).

If the cursor was on the last line then the screen will be scrolled provided that scrolling is enabled.

VDU 11

Vertical tab

Syntax

```
VDU 11
```

Parameters

—

Use

VDU 11 causes either the text cursor (by default) or the graphics cursor (in VDU 5 mode) to be moved back one line (ie in the negative Y direction). This normally means moving it up but will be different if the direction of cursor movement has been altered (using VDU 23,16).

If the cursor was on the first line then the screen will be scrolled, if scrolling is enabled.

VDU 12

Form feed/clear screen

Syntax

```
VDU 12
```

Parameters

—

Use

By default, VDU 12 clears either the current text window or, in VDU 5 mode, the current graphics window to the current text or graphics background colour respectively. The text or graphics cursor is moved to the text home position (see VDU 30).

When sent to a printer, this character generally causes a new page to be started.

VDU 13

Carriage return

Syntax

```
VDU 13
```

Parameters

—

Use

VDU 13 causes the text cursor or, in VDU 5 mode, the graphics cursor to be moved to the negative X edge of the relevant window at the same Y value. The negative X edge is normally the left edge but it may be changed using VDU 23,16.

When sent to a printer, this character generally causes the print head to move to the start of the current line. Additionally, some printers may also generate a line feed.

VDU 14

Page mode on

Syntax

```
VDU 14
```

Parameters

—

Use

VDU 14 causes the screen display to wait for Shift to be pressed before the next scroll and periodically thereafter. Normally, approximately 75% of the number of lines in the current window is scrolled before it waits again. The effects of the command may be cancelled using VDU 15.

OS_Byte 117 (page 2-153) may be used to determine whether page mode is enabled. See also OS_Byte 217 (page 2-175).

VDU 15

Page mode off

Syntax

```
VDU 15
```

Parameters

—

Use

VDU 15 cancels the effect of VDU 14 so that scrolling is unrestricted.

VDU 16

Clear graphics window

Syntax

```
VDU 16
```

Parameters

—

Use

VDU 16 clears the current graphics window to the current graphics background colour using the graphics background action. It does not affect the position of the graphics cursor.

VDU 17

Set text colour

Syntax

```
VDU 17,colour
```

Parameters

`colour`	logical text colour

Use

VDU 17 is used to assign a logical colour to either the text foreground or background according to the value of colour, as follows:

Value	**Colour**
0 - 127	foreground
128 - 255	background (colour in range 0 - 127)

If the absolute value of the parameter lies outside the allowed set for the current mode, it is treated MOD (the number of colours – 64 in 256 colour mode) so that it lies within that range. For example, in mode 1, which allows four colours, the commands VDU 17,9 and VDU 17,5 are equivalent to VDU 17,1.

The interpretation of colour depends on the type of mode:

Colours	**colour parameter meaning**		
2,4,16	Logical colour for that pixel		
256	Bottom 6 bits of colour provide colour information:		
		Bit 5	Blue High component
		Bit 4	Blue Low component
		Bit 3	Green High component
		Bit 2	Green Low component
		Bit 1	Red High component
		Bit 0	Red Low component

This allows 64 different colours to be obtained. Each of these can be used in one of four different tints, giving 256 available shades. See VDU 23,17 for more details. The current text colours may be read using OS_ReadVduVariables.

Example

`VDU 17,12`	*Set to logical colour 12*

VDU 18

Set graphics colour and action

Syntax

```
VDU 18,action,colour
```

Parameters

`action`	operation to perform
`colour`	colour to use

Use

VDU 18 is used to define either the graphics foreground colour or the graphics background colour, and the way in which it is to be plotted on the screen.

The graphics plotting action is determined by action as follows:

Value	**Action**
0	Overwrite colour on screen with colour
1	OR colour on screen with colour
2	AND colour on screen with colour
3	exclusive OR colour on screen with colour
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND colour on screen with (NOT colour)
7	OR colour on screen with (NOT colour)
8 - 15	As 0 to 7, but background colour is transparent
16 - 31	Colour pattern 1 using action 0 - 15
32 - 47	Colour pattern 2 using action 0 - 15
48 - 63	Colour pattern 3 using action 0 - 15
64 - 79	Colour pattern 4 using action 0 - 15
80 - 95	Giant colour pattern (patterns 1 - 4 placed side by side)

The range 8 - 15 is used in the following circumstances:

- If a sprite has a transparency mask, then plotting it using one of these actions causes the mask to be used.
- Where the mask has a 0 bit, nothing is plotted; where it has a 1 bit, the appropriate sprite colour is plotted. If an action in the range 0 - 7 is used, the sprite mask is ignored. See the chapter on sprites for more details.

These actions are also used in colour pattern plotting. If a pixel in the pattern has the same colour as the current graphics background colour, it is not plotted but left transparent instead. (If the action is used when setting a background colour pattern, then the pixel is left unplotted if it has the same colour as the current graphics foreground colour.)

The graphics colour is determined by colour as follows:

Value	**Meaning**
0 - 127	Foreground colour specified
128 - 255	Background colour specified (colour in range 0 - 127)

If the absolute value of the parameter lies outside the allowed set for the current mode, it is altered so that it lies within the range (as for VDU 17).

Where action has specified a colour pattern, then colour is used only to determine whether the pattern is used for the graphics foreground or background colour (depending on whether it is less than 128 or not).

The interpretation of colour depends on the type of screen mode. See the table for VDU 17 above for details.

The current graphics colours and actions may be read using OS_ReadVduVariables (page 2-211).

Example

`VDU 18,1,6`	*Write, ORing with the screen in colour 6*

VDU 19

Set palette

Syntax

`VDU 19,`*`logical colour,mode,red,green,blue`*

Parameters

`logical colour`	colour to set
`mode`	how to set the colour
`red, green, blue`	physical colour information

Use

VDU 19 defines the colour palette relationship. It causes a specified logical colour for either the screen, border or pointer to be represented by a given physical colour.

The action depends on the value of 'mode' as follows:

mode = 0 - 15	logical colour = actual colour red, green and blue are ignored
mode = 16	logical colour = red units red green units green blue units blue This sets both flash palettes for logical colour
mode = 17	Defines first flash palette for logical colour
mode = 18	Defines second flash palette for logical colour
mode = 24	Defines border colour = red units red green units green blue units blue logical colour is not used
mode = 25	Define logical colour (1 - 3) of pointer =

red units red
green units green
blue units blue

If you add 128 to the 'mode' value, you also set the 'supremacy' bit of the appropriate palette entry. This is used when the computers' video is mixed with an external video source, to provide a superimposed image.

In all cases, the red, green and blue parameters have a range 0 - 255. However, as only the top four bits are significant, the 16 possible values are &0X, &1X, &2X,... &FX, where X means 'don't care'. The bottom nibble may be significant in future versions of the hardware – to cater for this you should replicate the top nibble in the bottom nibble, by multiplying each RGB component by 17/16. Therefore, &F0F0F000 becomes &FFFFFF00.

In normal non-flashing colours, what this means is that both of the flash colours are the same. RISC OS will swap colours at a programmed interval. If they are the same colour, then there is no noticeable effect. 'Mode' values of 17 and 18 allow any colour to be made to flash with any combination of colours.

There are 16 palette registers, which means that in modes with one, two and four bits per pixel, there is a register available for each of the logical colours. Therefore, each can be assigned a physical colour by a simple one-to-one relationship.

By default (after a mode change or VDU 20), the palette is set up using a setting where the 'mode' value is in the range 0 - 15. The actual colour number depends on the logical colour and the number of bits per pixel used in a given screen mode as follows:

Logical	**Bits per pixel in a screen mode**		
colour	**1**	**2**	**4**
0	0	0	0
1	7	1	1
2		3	2
3		7	3
4			4
5			5
6			6
7			7
8			8
9			9
10			10
11			11
12			12
13			13
14			14
15			15

The meanings of the mode type colours are:

Physical colour	**Colour**
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White
8	Black-white flashing
9	Red-cyan flashing
10	Green-magenta flashing
11	Yellow-blue flashing
12	Blue-yellow flashing
13	Magenta-green flashing
14	Cyan-red flashing
15	White-black flashing

In modes with eight bits per pixel the situation is more complex. A simple mapping of the logical colour to the physical colour via the palette is not possible. Instead, the eight bits of the logical colour are treated as two nibbles as follows:

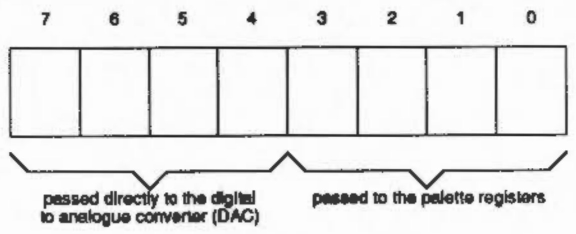

7
6
5
4
3
2
1
0
passed directly to the digital to analogue converter (DAC)
passed to the palette registers

Figure 21.1 Treatment of logical colours in eight bit per pixel modes

Bit 7	goes directly to the top bit of blue
Bit 6	goes directly to the top bit of green
Bit 5	goes directly to the second bit of green
Bit 4	goes directly to the top bit of red

The default palettes are set to have the following effect:

Bit 3	is sent to the second bit of blue
Bit 2	is sent to the second bit of red
Bit 1	is sent to the third bits of blue, green and red
Bit 0	is sent to the fourth bits of blue, green and red

Hence the palette can only be used to produce subtle effects upon the colour; it does not have any effect upon the top (most significant) bits of any colour or the second bit of green. It can only control the second bits of blue and red and the white tint which is obtained by the settings of all three of the third and fourth (least significant) bits.

You can also set the palette using OS_Word 12 (page 2-187), and read the current palette using OS_Word 11 (page 2-185) and OS_ReadPalette (page 2-209).

Example

```
VDU 19,5,12,0,0,0
```
Set logical colour 5 to be physical colour 12

VDU 20

Restore default colours

Syntax

```
VDU 20
```

Parameters

—

Use

VDU 20 restores the default palette for the current mode. It also resets the default text and graphics background colour to black, and the text and graphics foreground colour to white. The graphics foreground and background actions are set to 0 (overwrite). In 256-colour modes the tints are set to their default values (0 for background tints and &C0 for foreground ones).

VDU 21

Disable screen display

Syntax

```
VDU 21
```

Parameters

—

Use

VDU 21 prevents the VDU screen drivers performing any of their normal functions until a VDU 6 is issued. Any control sequences sent to the VDU drivers are queued in the usual way. Therefore, sending the character VDU 19 causes the next 5 characters to be treated as parameters for this (ignored) command.

For example, the sequence VDU 22,6 is treated as one whole command in the usual way and not as VDU 22 followed by VDU 6 which would re-enable the VDU drivers.

This command does not prevent characters from being sent to the VDU printer driver (if already enabled by a VDU 2), or any of the other output streams.

You can use OS_Byte 117 (page 2-153) to determine whether the VDU driver is currently enabled or disabled.

VDU 22

Change display mode

Syntax

```
VDU 22,mode
```

Parameters

mode	the screen mode to select

Use

VDU 22 is used to select a screen mode. The bottom seven bits of the mode parameter are used to select the mode. The modes available depend on the configured monitor type (see *Configure MonitorType on page 2-232) and the model of computer. Below is a table of all modes provided by RISC OS, which shows:

- the mode number
- the text resolution in columns × rows
- the graphics resolution in pixels, which corresponds to the clarity of the mode's display
- the resolution in OS units, which corresponds to the area of workspace shown by the mode
- the number of logical colours available
- the memory used per screen to the nearest 0.1Kbyte
- the vertical refresh rate to the nearest Hz (invalid for monitor type 5), which indicates the degree of flickering that you may perceive
- the bandwidth used to display the screen to the nearest 0.1Mbyte/second, which corresponds to the load the mode places on the computer
- the monitor types that support that mode:

Type	Monitor	
0	50Hz TV standard colour or monochrome monitor	
1	Multiscan monitor	
2	Hi-resolution 64Hz monochrome monitor	
3	VGA-type monitor	
4	Super-VGA-type monitor	(not available in RISC OS 2)
5	LCD (liquid crystal display)	(not available in RISC OS 2)

- the notes on the following page that are relevant to the mode.

Mode	Text resolution	Pixel resolution	OS units resolution	Logical colours	Mem used	Refresh rate	Band-width	Monitor types	Notes
0	80 × 32	640 × 256	1280 × 1024	2	20K	50Hz	1M/s	0,1,3,4,5	③
1	40 × 32	320 × 256	1280 × 1024	4	20K	50Hz	1M/s	0,1,3,4,5	③
2	20 × 32	160 × 256	1280 × 1024	16	40K	50Hz	2M/s	0,1,3,4,5	③
3	80 × 25	Text only	Text only	2	40K	50Hz	2M/s	0,1,3,4,5	③⑤⑦
4	40 × 32	320 × 256	1280 × 1024	2	20K	50Hz	1M/s	0,1,3,4,5	③
5	20 × 32	160 × 256	1280 × 1024	4	20K	50Hz	1M/s	0,1,3,4,5	③
6	40 × 25	Text only	Text only	2	20K	50Hz	1M/s	0,1,3,4,5	③⑤⑦
7	40 × 25	Teletext	Teletext	16	80K	50Hz	4M/s	0,1,3,4,5	③⑤
8	80 × 32	640 × 256	1280 × 1024	4	40K	50Hz	2M/s	0,1,3,4,5	③
9	40 × 32	320 × 256	1280 × 1024	16	40K	50Hz	2M/s	0,1,3,4,5	③
10	20 × 32	160 × 256	1280 × 1024	256	80K	50Hz	4M/s	0,1,3,4,5	③
11	80 × 25	640 × 250	1280 × 1000	4	39.1K	50Hz	2M/s	0,1,3,4,5	③⑧
12	80 × 32	640 × 256	1280 × 1024	16	80K	50Hz	4M/s	0,1,3,4,5	③
13	40 × 32	320 × 256	1280 × 1024	256	80K	50Hz	4M/s	0,1,3,4,5	③
14	80 × 25	640 × 250	1280 × 1000	16	78.2K	50Hz	3.9M/s	0,1,3,4,5	③⑧
15	80 × 32	640 × 256	1280 × 1024	256	160K	50Hz	8M/s	0,1,3,4,5	③
16	132 × 32	1056 × 256	2112 × 1024	16	132K	50Hz	6.6M/s	0,1	⑥
17	132 × 25	1056 × 250	2112 × 1000	16	129K	50Hz	6.5M/s	0,1	⑥⑧
18	80 × 64	640 × 512	1280 × 1024	2	40K	50Hz	2M/s	1	
19	80 × 64	640 × 512	1280 × 1024	4	80K	50Hz	4M/s	1	
20	80 × 64	640 × 512	1280 × 1024	16	160K	50Hz	8M/s	1	
21	80 × 64	640 × 512	1280 × 1024	256	320K	50Hz	16M/s	1	
23	144 × 56	1152 × 896	2304 × 1792	2	126K	64Hz	8.1M/s	2	
24	132 × 32	1056 × 256	2112 × 1024	256	264K	50Hz	13.2M/s	0,1	⑥
25	80 × 60	640 × 480	1280 × 960	2	37.5K	60Hz	2.3M/s	1,3,4,5	④
26	80 × 60	640 × 480	1280 × 960	4	75K	60Hz	4.5M/s	1,3,4,5	④
27	80 × 60	640 × 480	1280 × 960	16	150K	60Hz	9M/s	1,3,4,5	④
28	80 × 60	640 × 480	1280 × 960	256	300K	60Hz	18M/s	1,3,4,5	④
29	100 × 75	800 × 600	1600 × 1200	2	58.6K	56Hz	3.3M/s	1,4	①②
30	100 × 75	800 × 600	1600 × 1200	4	117.2K	56Hz	6.6M/s	1,4	①②
31	100 × 75	800 × 600	1600 × 1200	16	234.4K	56Hz	13.2M/s	1,4	①②
33	96 × 36	768 × 288	1536 × 1152	2	27K	50Hz	1.4M/s	0,1	①
34	96 × 36	768 × 288	1536 × 1152	4	54K	50Hz	2.7M/s	0,1	①
35	96 × 36	768 × 288	1536 × 1152	16	108K	50Hz	5.4M/s	0,1	①
36	96 × 36	768 × 288	1536 × 1152	256	216K	50Hz	10.8M/s	0,1	①
37	112 × 44	896 × 352	1792 × 1408	2	38.5K	60Hz	2.3M/s	1	①
38	112 × 44	896 × 352	1792 × 1408	4	77K	60Hz	4.6M/s	1	①
39	112 × 44	896 × 352	1792 × 1408	16	154K	60Hz	9.2M/s	1	①
40	112 × 44	896 × 352	1792 × 1408	256	308K	60Hz	18.5M/s	1	①
41	80 × 44	640 × 352	1280 × 1408	2	27.5K	60Hz	1.7M/s	1,3,4,5	①③④
42	80 × 44	640 × 352	1280 × 1408	4	55K	60Hz	3.3M/s	1,3,4,5	①③④
43	80 × 44	640 × 352	1280 × 1408	16	110K	60Hz	6.6M/s	1,3,4,5	①③④
44	80 × 25	640 × 200	1280 × 800	2	15.7K	60Hz	0.9M/s	1,3,4,5	①③
45	80 × 25	640 × 200	1280 × 800	4	31.3K	60Hz	1.9M/s	1,3,4,5	①③
46	80 × 25	640 × 200	1280 × 800	16	62.5K	60Hz	3.8M/s	1,3,4,5	①③

Notes on display modes

1. These modes are not available in RISC OS 2.00, nor (except for mode 31) are they available in RISC OS 2.01.
2. These modes are not available on early models of RISC OS computers (ie the Archimedes 300 and 400 series, and the A3000), because they are unable to clock VIDC at the necessary rate.
3. These modes are handled differently with a VGA or Super-VGA-type monitor. If you are using such a monitor:
   - RISC OS 2.00 does not implement these modes.
   - The picture is displayed on a screen having 352 raster lines. Where a mode has fewer than 352 vertical pixels, it is centred on the screen with blank rasters at the top and bottom. Because of their appearance these modes are known as *letterbox modes*.
   - The refresh rate is 70Hz.
   - The bandwidths shown in the table for these modes are lower than these monitor types consume, because no allowance has been made for the blank rasters.
   - Early models of RISC OS computers (ie the Archimedes 300 and 400 series, and the A3000) scan these modes some 4.7% slow. Again this is because they are unable to clock VIDC at the necessary rate. Most VGA and Super-VGA-type monitors can still successfully lock onto this signal, but some may not. Furthermore, these models do not provide a *Sync Polarity* signal. This makes the effect of *letterbox modes* (see above) more severe.
4. Early models of RISC OS computers (ie the Archimedes 300 and 400 series, and the A3000) scan these modes some 4.7% slow with any type of monitor. Again this is because they are unable to clock VIDC at the necessary rate.
5. These modes do not display graphics, for compatibility with BBC/Master series computers.
6. In these modes circles, arcs, sectors and segments do not look circular. This is because the aspect ratio of the pixels is not in a 1:2, 1:1 or 2:1 ratio.
7. This is a *gap mode*, where the colour of the gaps is not necessarily the same as the text background.
8. These modes are not a multiple of eight pixels high. By default, in these modes the bottom of the screen corresponds to the bottom line of ECF patterns, but the top line will not correspond to the top line of ECF patterns.

Modes 22 and 32 have not been defined.

If an attempt is made to select a mode which is not appropriate to the current monitor type (or OS version), a suitable mode for that monitor is used. For example, an attempt to select mode 23 on a type 0 monitor will result in mode 0 being used.

In 256 colour modes, there are some restrictions on the control of the colours. Only 64 base colours may be selected; 4 levels of tinting turn the base colours into 256 shades. Also, the selection from the colour palette of 4096 shades is only possible in groups of 16.

Banks of screen memory

If 128 is added to the mode number, the so-called shadow bank is used if possible. Any display mode may have several banks of memory available. The number of banks depends on the size of the screen memory (as allocated by *Configure ScreenSize) and the size of the current mode. For example, if 160K is allocated, and 20K is used for the display, eight banks are available.

Usually, bank 1 is used. However, if 128 is added to the mode number, or a *Shadow command has been issued, bank 2 is used after a mode change. Shadow memory can only be used if ScreenSize is at least twice the memory for the required mode.

The other banks may be accessed using OS_Bytes 112 - 113.

Effect of the mode command

The mode command causes the following actions:

- Cursor editing is terminated if currently in use
- VDU 4 mode is entered
- The text and graphics windows are restored to their default values
- The text cursor is moved to its home position
- The graphics cursor is moved to (0,0)
- The graphics origin is moved to (0,0)
- Paged mode is terminated if currently in use
- The logical-physical colour map is set to the new mode's default
- The text and graphics foreground colours are set to white
- The text and graphics background colours are set to black (colour 0)
- The colour patterns are set to their defaults for the new mode
- The ECF origin is set to (0,0)
- The dot pattern for dotted lines is reset to &AAAAAAAA

- The dot pattern repeat length is reset to 8
- The screen is cleared to the current text background colour (ie black).

If there is not enough configured screen RAM for the mode you have selected, and the application workspace area is not in use, then memory is moved out of the application workspace area to the screen area.

Getting information on a mode

The current screen mode may be read using OS_Byte 135 (page 2-157).

The size of the screen in a given mode can be determined by reading VDU variables XWindLimit, YWindLimit, XEigFactor, YEigFactor.

Example

```
VDU 22,7                    Select Teletext mode
```

VDU 23

Miscellaneous commands

Syntax

```
VDU 23,command,n1,n2,n3,n4,n5,n6,n7,n8
```

Parameters

command	the command to perform
n1,n2,n3,n4,n5,n6,n7,n8	the 8 parameters which follow it

Use

VDU 23 is a multi-purpose command taking nine parameters, of which the first identifies a particular function. Each of the available functions is described below. Eight additional parameters are required in each case, though often most of these are ignored. This enables you to use '|' as shorthand in VDU statements, eg:

Examples

```
VDU 23,0,10|                      These two lines have the same effect
VDU 23,0,10,0,0,0,0,0,0,0
```

VDU 23,0

Set the interlace and controls cursor appearance

Syntax

```
VDU 23,0,action,mode,0,0,0,0,0,0
```

Parameters

action	Sets which action to perform
mode	Defines the mode for a given action

Use

If *action*= 8, this sets the interlace as follows:

Mode	Effect
0	sets the screen interlace state to the opposite of the current *TV setting
1	sets the screen interlace state to the current *TV setting
&80	turns the screen interlace off
&81	turns the screen interlace on

If *action*= 10 or 11, this controls the height of the cursor on the screen and its appearance.

action= 10 *mode* defines the start line for the cursor and its appearance:

Bits 0 - 4 define the start line (0 being the top)
Bits 5 - 6 define its appearance as follows:

Bit 6	Bit 5	Meaning
0	0	Steady
0	1	Off
1	0	Fast flash
1	1	Slow flash

action= 11 *mode* defines the end line for the cursor.

The bottom line of the cursor is 7 for 'normal' modes, 9 for standard 'gap' modes, and 19 for mode 7.

Example

```
VDU 23,0,8,&81|          Turn the screen interlace on
```

VDU 23,1

Control the appearance of the cursor

Syntax

```
VDU 23,1,mode,0,0,0,0,0,0
```

Parameters

mode	determines which cursor mode

Use

VDU 23,1 controls the appearance of the cursor on the screen depending on the value of *mode*:

Value	**Meaning**
0	stops the cursor appearing
1	makes the cursor re-appear
2	makes the cursor steady
3	makes the cursor flash

The effect of this call is cancelled when cursor editing occurs. The effect of the previous call is not changed by cursor editing. See also SWI OS_RemoveCursors and SWI OS_RestoreCursors.

Example

```
VDU 23,1,3|
```
makes the cursor flash

VDU 23,2-5

Define ECF pattern and colours

Syntax

```
VDU 23,2|3|4|5,n1,n2,n3,n4,n5,n6,n7,n8
```

Parameters

n1,n2,n3,n4,n5,n6,n7,n8	colour for each row

Use

VDU 23,2 - VDU 23,5 are used to define the four colour patterns:

VDU 23,2	sets pattern 1
VDU 23,3	sets pattern 2
VDU 23,4	sets pattern 3
VDU 23,5	sets pattern 4

Each of the integers n1 to n8 defines the colours of one row of the pattern, n1 being the top row and n8 being the bottom. For a given parameter, the logical colours of the pixels in each row depend upon the number of colours available in the current screen mode and which pattern mode is used. There are two available pattern modes. The default is the BBC/Master compatible mode. The other is the native RISC OS mode which decodes the values in a simpler fashion. To change between these modes use VDU 23,17,4.

If the bit settings in one of the n parameters is denoted by 76543210, then the logical colours of the pixels in each row (from left to right) are:

Bits per pixel	**No. of colours**	**No. of pixels in a line**	**BBC/Master colours**	**RISC OS colours**
1	2	8	7,6,5,4,3,2,1,0	0,1,2,3,4,5,6,7
2	4	4	73, 62, 51, 40	10, 32, 54, 76
4	16	2	7531, 6420	3210, 7654
8	256	1	76543210	76543210

There are many examples of using these and the VDU 23,12-15 commands to alter ECF functions in the section entitled *Application Notes* on page 2-243.

In any 256 colour mode, each parameter refers to the colour of each line. Use the colour byte as described by VDU 19 (page 2-79).

VDU 23,6

Set dot-dash line style

Syntax

```
VDU 23,6,n1,n2,n3,n4,n5,n6,n7,n8
```

Parameters

n1,n2,n3,n4,n5,n6,n7,n8	bit pattern for style

Use

VDU 23,6 sets the dot-dash line style used by dotted line PLOT commands (see also VDU 25 on page 2-118, and OS_Byte 163 on page 2-162).

Each of the integers n1 to n8 defines eight elements of the line style, n1 being at the start and n8 at the end. The bits in each byte are read from most significant to least significant, each 1-bit indicating a dot and each 0-bit a space. The default is &AAAAAAAA (alternating dots and spaces) with a repeat length of eight (so only n1 is used).

Example

```
VDU 23,6,&F0,&F0,&F0,&F0,&F0,&F0,&F0,&F0
```

VDU 23,7

Scroll text window or screen

Syntax

```
VDU 23,7,extent,direction,movement,0,0,0,0,0
```

Parameters

extent	text window or screen
direction	direction to scroll
movement	how much movement

Use

VDU 23,7 allows the current text window or whole screen to be scrolled directly in any direction without moving the cursor. The extent, direction and movement determine the area to be scrolled, the direction of scrolling and the amount of scrolling as follows:

extent	**Effect**
0	scroll the current text window
1	scroll the entire screen

direction	**Effect**
0	scroll right
1	scroll left
2	scroll down
3	scroll up
4	scroll in positive X direction
5	scroll in negative X direction
6	scroll in positive Y direction
7	scroll in negative Y direction

movement	**Effect**
0	scroll by one character cell
1	scroll by one character cell vertically or one byte horizontally

If movement is 1, the horizontal movement depends on the number of colours in the current mode as follows:

Number of colours	Number of pixels moved
2	8
4	4
16	2
256	1

Example

```
VDU 23,7,0,3,0|          Scroll window up one character
```

VDU 23,8

Clear a block of the text window

Syntax

VDU 23,8,*base_start*,*base_end*,*x1*,*y1*,*x2*,*y2*,0,0

Parameters

base_start	base position of start of bloc
base_end	base position of end of block
x1, y1, x2, y2	displacements of block

Use

VDU 23,8 causes a block of the current text window to be cleared to the text background colour. The parameters base start and base end indicate base positions relating to the start and end of the block to be cleared respectively:

Value	Meaning
0	top left of window
1	top of cursor column
2	off top right of window
4	left end of cursor line
5	cursor position
6	off right of cursor line
8	bottom left of window
9	bottom of cursor column
10	off bottom right of window

References to 'left', 'up' and so on are dependent upon the cursor movement control set by VDU 23,16. 'Off' means 'one character beyond (in the positive x direction)'. The effects of other values, ie. 3, 7 and any number over 10, are undefined.

The parameters x1,y1 and x2,y2 are displacements from the positions specified by the base start and base end; they determine the start and end of the block:

x1	Displacement from base start in x direction
y1	Displacement from base start in y direction
x2	Displacement from base end in x direction
y2	Displacement from base end in y direction

The result is undefined if the absolute values defining the start and end of the block produce values outside the range –128 to 127. If the end point of the block lies before the start point then no clearing takes place.

The action of this command can be viewed as equivalent to moving the text cursor to the start of the block, then printing spaces until the end of the block is reached (but without printing a space at the last position).

Example

`VDU 23,8,5,10,0,0,0,0|`	*Clear from cursor to end of screen*

VDU 23,9

Set flash time for first flashing colour

Syntax

```
VDU 23,9,duration,0,0,0,0,0,0,0
```

Parameters

duration	number of VSyncs

Use

VDU 23,9 sets the flash time for the first flashing colour. The length is determined by the value of duration as follows:

duration = 0	sets a steady flash colour 1
duration ≠ 0	sets the duration

A Vsync is the time between refreshes of the screen display. It varies between display modes and countries. In the UK for modes 0 - 17 it is approximately 1/50th second.

This command is equivalent to OS_Byte 9 (see page 2-136).

Example

`VDU 23,9,1|`	*Set to one Vsync*

VDU 23,10

Set flash time for second flashing colour

Syntax

```
VDU 23,10,duration,0,0,0,0,0,0,0
```

Parameters

duration	number of VSyncs

Use

VDU 23,10 sets the flash time for the second flashing colour. The length is determined by the value of duration as follows:

duration = 0	sets a steady flash colour 2
duration ≠ 0	sets the duration

This command is equivalent to OS_Byte 10 (page 2-138).

Example

`VDU 23,10,2|`	Set to two VSyncs

VDU 23,11

Set default patterns

Syntax

```
VDU 23,11,0,0,0,0,0,0,0,0
```

Parameters

—

Use

VDU 23,11 selects the Master 128 compatible pattern mode and causes the four colour patterns to be reset to their defaults for the current screen mode. With the default logical-physical map, these defaults are:

Mode 0

1 – Red-orange	2 – Orange	3 – Yel-orange	4 – Cream
11001100	11110000	11111111	00000011
00000000	00001111	00110011	00001100
11001100	11110000	11111111	01000100
00000000	00110011	01010101	10001000

Modes 1,5,8,11,19,26

1 – Red-orange	2 – Orange	3 – Yel-orange	4 – Cream
2121	2121	2222	2323
1111	1212	1212	3232
2121	2121	2222	2323
1111	1212	1212	3232

Modes 2,9,12,14,16,17,20,27

1 – Orange	2 – Pink	3 – Yel-green	4 – Cream
21	61	32	37
12	16	23	73
21	61	32	37
12	16	23	73

Modes 4,18,23,25

1 – Dark grey	2 – Grey	3 – Light grey	4 – Hatching
10101010	11001100	11111111	00010001
00000000	00110011	01010101	00100010
10101010	11001100	11111111	01000100
00000000	00110011	01010101	10001000

Modes 10,13,15,21,24,28

1 – Grey		2 – Slate		3 – Green		4 – Pink	
3F	00	0	C0	4	C0	3B	00
	40		80		80		40
	80		40		40		80
	C0		00		00		C0

All the patterns repeat after four rows, so only the first four are shown.

Example

```
VDU 23,11|
```

VDU 23,12-15

Define simple ECF patterns and colours

Syntax

```
VDU 23,pattern,n1,n2,n3,n4,n5,n6,n7,n8
```

Parameters

Define a two by four block of pixels as follows:

n1	n2
n3	n4
n5	n6
n7	n8

The pattern parameter determines which colour pattern is set:

pattern	Sets colour pattern
12	1
13	2
14	3
15	4

Use

VDU 23,12-15 are used to define the four colour patterns in a simpler way than that provided by VDU 23,2-5. The limitation is that you can only set a two-by-four pattern of pixels.

The pixels of the top row of the resulting pattern are assigned alternating logical colours n1 and n2, those of the next row have colours n3 and n4 etc.

In any 256 colour mode, the declared use of the parameters does not apply. In this case, each parameter refers to the colour of each line, from 1 to 8. Use the colour byte as described by VDU 19 (page 2-79).

Example

To set up the following pattern in mode 1 for colour pattern 1:

RedYel	12
WhtRed	31
BlkRed	01
WhtYel	32

the required sequence is:

```
VDU 23,12,1,2,3,1,0,1,3,2
```

VDU 23,16

Control the movement of cursor after printing

Syntax

```
VDU 23,16,x,y,0,0,0,0,0,0
```

Parameters

x	exclusive OR value
y	AND value

Use

VDU 23,16 gives control of the movement of the cursor after a character has been printed. This movement is under the control of a byte of flags. VDU 23,16 replaces the byte by:

((current byte) AND y) XOR x

The interpretation of the flags is as follows:

Bit	Value	Effect
7	0	Normal.
	1	Undefined.
6	0	In VDU 5 mode, cursor movements beyond the current edge of the window cause special actions. For example, they generate newlines at the end of the line.
	1	In VDU 5 mode, cursor movements beyond the edge of the window do not cause special actions. This is the most useful mode of VDU 5; used in the Window Manager.
5	0	Cursor moves in the positive X direction after the character is printed. If this results in the cursor moving beyond the edge of the window, the settings of bits 6, 4 and 0 define the action which is taken.
	1	Cursor does not move after the character is printed.
4	0	When a cursor movement in the Y direction results in the cursor moving beyond the window edge, the window is scrolled if in VDU 4 mode. If in VDU 5 mode, the cursor moves to the opposite edge of the window.
	1	When a cursor movement in the Y direction results in the cursor moving beyond the window edge, the cursor is always moved to the opposite edge of the window.

3	0	X direction is horizontal, Y direction is vertical.
	1	X direction is vertical, Y direction is horizontal.
2	0	Positive vertical direction is down.
	1	Positive vertical direction is up.
1	0	Positive horizontal direction is right.
	1	Positive horizontal direction is left.
0	0	Disables the scroll-protect option. When printing a character in VDU 4 mode results in the cursor moving beyond the edge of the window, the cursor is instead moved to the negative X edge of the window and one line in the positive Y direction.
	1	Enables the scroll protect option. When printing a character in VDU 4 mode results in the cursor moving beyond the edge of the window, a 'pending newline' is generated. It is actually executed just before the next character is printed, provided that it has not been deleted or executed by another cursor control character. For example VDU 127 would cancel it; VDU 9 would execute it.

Example

```
VDU 23,16,%00000100,%11111011|          Set vertical direction down
```

VDU 23,17,0-3

Set the tint for a colour

Syntax

```
VDU 23,17,action,tint,0,0,0,0,0,0
```

Parameters

action	determines which colour is to be set
tint	what the tint is to be set to

Use

VDU 23,17,0-3 is used to set the tint for a colour in the 256-colour modes. The action determines which colour is set, as follows:

action	Colour
0	sets the tint for the text foreground colour
1	sets the tint for the text background colour
2	sets the tint for the graphics foreground colour
3	sets the tint for the graphics background colour

This command controls the top two bits of blue, green and red independently of each other, and also allows the bottom two bits to be controlled. However, they cannot be set independently. The least significant bits must either all be set or all clear. Hence it determines the amount of white tint given to the colour. The value of the tint is given by the top two bits of tint:

tint	Tint effect
&00	Bit 0 and bit 1 clear (darkest)
&40	Bit 0 set and bit 1 clear
&80	Bit 1 set and bit 0 clear
&C0	Bit 0 and bit 1 set (lightest)

When a pixel is plotted the following occurs, in terms of the actual logical colour stored in the screen memory: the bottom six bits of the colour number (set by VDU 17-18) are moved to bits 2 - 7 of the colour byte, and their order is changed; the appropriate tint value is shifted down by six bits, into bits 0 and 1, and the two parts are then combined.

Example

```
VDU 23,17,0,&C0|          Set the text foreground colour to lightest tint
```

VDU 23,17,4

Choose the patterns used to interpret subsequent VDU 23,2 - 5... calls

Syntax

```
VDU 23,17,4,patterns,0,0,0,0,0,0
```

Parameters

patterns	which mode of patterns

Use

This command chooses which set of colour patterns are used to interpret subsequent VDU 23,2 - 5... calls, depending on the value of <patterns>:

patterns	**Mode**
0	Use 6502 BBC Micro compatible colour patterns
1	Use native colour patterns

Example

```
VDU 23,17,4,1|            Use native colour patterns
```

VDU 23,17,5

Exchange text foreground and background colours

Syntax

```
VDU 23,17,5,0,0,0,0,0,0,0
```

Parameters

—

Use

This command exchanges the current text foreground and background colours. After the first time it's called, subsequent characters printed are in inverse video. After the second time it's called, subsequent characters printed are of normal appearance.

Example

```
VDU 23,17,5|
```

VDU 23,17,6

Set ECF origin

Syntax

```
VDU 23,17,6,x;y;0,0,0
```

Parameters

x, y	point coordinates

Use

By default, the alignment of ECF patterns is with the bottom left corner of the screen. This command changes it so that the bottom left of the pattern coincides with the bottom left of the specified point.

The origin is restored to the default after a mode change.

OS_SetECFOrigin (page 2-226) performs the same action.

Example

```
VDU 23,17,6,200;300;0,0,0
```

VDU 23,17,7

Set character size/spacing

Syntax

```
VDU 23,17,7,flags,x;y;0,0
```

Parameters

flags	what to set the size of
x, y	size in pixels

Use

This command allows changing the size and spacing of VDU 5 characters. They are reset when a mode change occurs. Bit 1 of the flags refers to the size of VDU 5 characters. Bit 2 refers to the spacing between VDU 5 characters. x and y are sizes in pixels.

Sizes of 8x16 and 8x8 are optimised for speed. All other settings are much slower. The spacing settings do not affect the speed. The default size and spacing of VDU 5 characters is 8x8.

Example

```
VDU 23,17,7,%100,10;8;0,0          change VDU 5 spacing to 10 pixels
```

VDU 23,18-24

Reserved for future expansion

VDU 23,25-26

These calls are provided by the Font Manager for compatibility with earlier operating systems. You must not use them. See the chapter entitled *The Font Manager* on page 5-1 for further details.

VDU 23,27

This call is provided by the Sprite Manager. See the chapter entitled *Sprites* on page 2-247 for further details.

VDU 23,28-31

Reserved for use by application programs.

VDU 23,32-255

Redefine the printable characters

Syntax

```
VDU 23,32-255,n1,n2,n3,n4,n5,n6,n7,n8
```

Parameters

32 – 255	character to define
n1, n2, n3, n4, n5, n6, n7, n8	definition by row

Use

VDU 23,32 to VDU 23,255 redefine the printable ASCII characters. The redefined character depends on the value of the second parameter. For example, VDU 23,65 redefines the character whose ASCII value is 65, ie capital A. The parameters n1 to n8 are integers representing the eight rows of the character to be redefined, n1 being the top row and n8 the bottom row. Each bit of a value represents one pixel of the corresponding row, with a '1' indicating that the corresponding pixel is to be plotted in the foreground colour and a '0' that it is to be plotted in the background colour (or not at all in the case of VDU 5 mode printing). The most significant bit of the byte corresponds to the left-hand pixel of its row, and the others follow linearly.

Although the delete character (ASCII 127) can be redefined, redefining has no effect as it cannot be displayed.

You can read the pattern for a given character using OS_Word 10 (page 2-183).

You should not use this call in programs that might be run under the desktop, as your redefinitions may affect other programs. If you must use this call, ensure you only redefine characters that are normally unused.

Note that the desktop redefines some characters for its own use, and you must not redefine these yourself. To determine which characters are normally unused, view the entire system font under the desktop (the !Chars application is ideal for this):

- In RISC OS 2, the unused characters are those remaining from the underlined string 'These·characters·are·not·defined'
- In later versions of RISC OS, the unused characters are those represented as small hexadecimal numbers

Example

```
VDU 23,65,&AA,&55,&AA,&55,&AA,&55,&AA,&55          redefine 'A'
```

VDU 24

Define graphics window

Syntax

```
VDU 24,x1;y1;x2;y2;
```

Parameters

x1, y1, x2, y2	coordinates of window

Use

VDU 24 allows the user to define a graphics window. Any graphics objects which are drawn (including VDU 5 mode and fancy-font characters) and which lie outside this window are clipped to the edges of the window. The four parameters define the left, bottom, right and top boundaries of the window respectively, relative to the current graphics origin (the bottom left of the screen, by default). The window which you are defining must lie within the screen boundaries, otherwise the command is ignored.

The coordinates are inclusive – that is, the points you specify lie within the window.

Use OS_ReadVduVariables (page 2-211) to discover the size of the current graphics window.

Example

```
VDU 24,100;150;700;800;
```

The following example shows how to derive (in this instance, xsize) the size of a window in OS units

```
DIM blk% 12
VduExt_XEigFactor% = 4
VduExt_XWindLimit% = 11
blk%!0 = VduExt_XEigFactor%
blk%!4 = VduExt_XWindLimit%
blk%!8 = -1
SYS "OS_ReadVduVariables", blk%, blk%
xeigfactor% = blk%!0
xwindlimit% = blk%!4: REM in pixels
xwindsize%  = (xwindlimit% + 1) << xeigfactor%: REM in OS units
```

VDU 25

General PLOT command

Syntax

```
VDU 25,type,x;y;
```

Parameters

type	what kind of plot to perform
x, y	where to plot

Use

VDU 25 is a multi-purpose graphics plotting command. The first parameter defines a particular function. The other parameters are the x coordinate and the y coordinate. They are relative either to the current graphics origin, or to the last point visited, depending on the value of type.

The bottom three bits of type determine the manner in which the plot is to be performed. Thus (type AND 7) has the following effects:

type AND 7	**Effect**
0	move cursor relative (to last graphics point visited)
1	plot relative using current foreground colour
2	plot relative using logical inverse colour
3	plot relative using current background colour
4	move cursor absolute (ie move to actual coordinates given)
5	plot absolute using current foreground colour
6	plot absolute using logical inverse colour
7	plot absolute using current background colour

The remaining bits of type determine the action to be performed. The value given here is added to the 0 - 7 range above to get all the possible combinations. The value here is the decimal starting value:

Value	**Effect**
0	Solid line including both end points
8	Solid line excluding the final point
16	Dotted line including both endpoints, pattern restarted
24	Dotted line excluding the final point, pattern restarted
32	Solid line excluding the initial point
40	Solid line excluding both end points
48	Dotted line excluding the initial point, pattern continued
56	Dotted line excluding both end points, pattern continued
64	Point Plot
72	Horizontal line fill (left and right) to non-background
80	Triangle fill
88	Horizontal line fill (right only) to background
96	Rectangle fill
104	Horizontal line fill (left and right) to foreground
112	Parallelogram fill
120	Horizontal line fill (right only) to non-foreground
128	Flood to non-background
136	Flood to foreground
144	Circle outline
152	Circle fill
160	Circular arc
168	Segment
176	Sector
184	Block copy/move *
192	Ellipse outline
200	Ellipse fill
208	Font printing – see the chapter entitled *The Font Manager*
216	Reserved for Acorn Expansion
224	Reserved for Acorn Expansion
232	Sprite Plot – see the chapter on sprites
240	Reserved for User programs
248	Reserved for User programs

* The eight values in the range 184 - 191, which perform a block copy/move, have the following meanings:

Value	**Effect**
184	Move relative
185	Relative rectangle move
186	Relative rectangle copy
187	Relative rectangle copy
188	Move absolute
189	Absolute rectangle move
190	Absolute rectangle copy
191	Absolute rectangle copy

Some of the objects require several points to be specified in order to define the shape completely. The last plot does the actual drawing. The sequences of moves and draws required for each type are:

Shape	**Sequence of moves**
Line	Move to one endpoint. Plot line to other endpoint.
Triangle	Move to first vertex. Move to second vertex. Plot triangle to last vertex.
Rectangle	Move to one corner. Plot rectangle to diagonally-opposite corner.
Parallelogram	Move to first corner. Move to second corner. Plot parallelogram to third corner. The fourth corner is derived from the other three, and is opposite the second one.
Circle	Move to centre. Plot circle to point on the circumference.
Arc, segment, sector	Move to centre of circle. Move to start of arc. Plot to a point on the line from the centre to the end of the arc. Arcs, etc, are always drawn counter-clockwise.
Block copy/move	Move to one corner of source rectangle. Move to diagonally-opposite corner of source rectangle. Plot block copy/move to lower left of destination rectangle.
Ellipse	Move to centre. Move to intersection of ellipse circumference and centre's Y coordinate. Plot ellipse to highest or lowest point on the ellipse.

Example

```
VDU 25,69,100;200;
```
plot point absolute

VDU 26

Restore default windows

Syntax

```
VDU 26
```

Parameters

—

Use

VDU 26 causes the text and graphics windows to be reset to their default states, ie both become the full screen. In addition, the command resets the graphics origin to (0,0), moves the graphics cursor to (0,0) and moves the text cursor to its home position. Hardware scrolling of the text window is initiated.

VDU 27

No operation

Syntax

```
VDU 27
```

Parameters

—

Use

This VDU has no effect

VDU 28

Define text window

Syntax

```
VDU 28,x1,y1,x2,y2
```

Parameters

x1	left-most x column
y1	bottom-most y row
x2	right-most x column
y2	top-most y row

Use

VDU 28 defines (or redefines) a text window. The parameters are integers specifying the boundary of the window as above.

If the command attempts to define a window which extends outside the screen boundaries, has x1 greater than x2, or has y1 less than y2, it will have no effect. The smallest possible window is one character.

You can read the size of the current text window using OS_ReadVduVariables (page 2-211).

Example

```
VDU 28,10,15,20,5
```

VDU 29

Set graphics origin

Syntax

```
VDU 29,x;y;
```

Parameters

x, y	where the origin is to be set

Use

VDU 29 defines the point specified as the origin to be used for all subsequent graphics output using VDU 25 commands, and for the graphics window defined by VDU 24. The parameters are the two pairs of bytes specifying the absolute x and y coordinates of the new origin.

- Note: changing the graphics origin does not alter the position of the graphics window on the screen. The window's coordinates in terms of the origin therefore effectively change after a VDU 29.

You can read the position of the current origin using OS_ReadVduVariables (page 2-211).

Example

```
VDU 29,100;200;
```

VDU 30

Home text cursor

Syntax

```
VDU 30
```

Parameters

—

Use

VDU 30 moves the text cursor to its 'home' position. This is normally the top left of the window but may be changed (using VDU 23,16). In VDU 5 mode the graphics cursor is moved instead. It may have an offset of up to (character size −1) pixels out of the corner along one or both of the axes to allow for the height or width of the character depending on the direction of character printing.

VDU 31

Position text cursor

Syntax

```
VDU 31,x,y
```

Parameters

x, y	text position to move to

Use

VDU 31 moves the text cursor to a specified x and y coordinate on the screen. The parameters x and y are the column and row numbers.

In VDU 4 mode, x and y are given relative to the text 'home' position which is at (0,0). If the position lies outside the text window, nothing happens, unless the scroll protect option is enabled and the x coordinate is just beyond the positive X edge of the window. In this case, the text cursor is moved to position (x−1,y) and a pending newline is generated.

In VDU 5 mode the graphics cursor is moved to its 'home' position plus
(x character spacing * x) pixels in the positive X direction, plus
(y character spacing * y) pixels in the positive Y direction. It is possible to move the cursor outside the graphics window in VDU 5 mode.

You can read the position of the text cursor using OS_Byte 134 (page 2-155).

Example

```
VDU 31,10,5
```

VDU 127

Delete

Syntax

```
VDU 127
```

Parameters

—

Use

Unless the previous use of VDU 23,16 indicates that no cursor movement is to take place after character printing, the cursor is moved backwards as if by VDU 8. Then the character under the cursor is deleted by overprinting it with a space (in VDU 4 mode) or a solid block of graphics background colour (in VDU 5 mode). These space and solid block characters are selected from the 'hard' (rather than the 'soft') font, so redefining these characters will not change the results.

Service Calls

Service_ModeChange (Service Call &46)

Mode change

On entry

R1 = &46 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is made whenever a mode change has taken place. It is made for the benefit of modules which may want to re-read some VDU variables to keep a consistent view of the world. It should not be claimed; there is nothing a module can do to prevent the mode change from taking place.

Service_PreModeChange (Service Call &4D)

Mode change

On entry

R1 = &4D (reason code)
R2 = selected mode (before possible translation)

On exit

Case 1

R1 preserved
R2 preserved

This is the normal action for a module which does not want to interfere

Case 2

R1 = 0 (service claimed)
R0 = 0

This implies that the module does not want the mode change to take place, and has taken an alternative action.

Case 3

R1 = 0 (service claimed)
R0 pointer to an error block

This implies that the module does not want the mode change to take place, and wishes to return the error pointed to by R0.

Case 4

R1 preserved
R2 = new mode

This implies that the module wants to substitute a mode for the specified mode. This is not a very good way of doing it, as other modules further down the chain will be offered the service with this new mode. The Service_ModeTranslation mechanism described above should be used by modules providing new monitor types.

Use

In RISC OS it is possible to load modules which provide additional screen modes and additional monitor types. This service call is used for mode change requests.

Service_ModeExtension (Service Call &50)

Allow soft modes

On entry

R1 = &50 (reason code)
R2 = mode number that information is requested for
R3 = monitor type (or -1 for don't care)

On exit

All registers preserved (if not claimed)

If claimed:
R1 = 0
R2 preserved
R3 = pointer to VIDC list
R4 = pointer to workspace list

Use

In RISC OS it is possible to load modules which provide additional screen modes and additional monitor types.

All values are words in the VIDC list; its format is:

Offset	Value
0	0 (indicates format of list, to allow for new VIDCs at a later date)
4	VIDC base mode
8	VIDC parameter
12	VIDC parameter
...	...
n	–1

The VIDC base mode is the number of an existing operating system screen mode which is used to determine the values of VIDC registers not explicitly mentioned in the list. The VIDC parameters are in the form that would be written to the hardware: ie the top 6 bits specify which register is programmed, and the remainder specify the value to be programmed in that register.

However, bits 6 and 7 of the control register should be set to 0, as these will be modified by RISC OS to take the configured sync and the *TV interlace setting into account. Similarly the vertical parameters for border start, display start, display end and border end are modified by RISC OS to take the *TV vertical offset into account.

VIDC parameters below &80000000 are ignored, since these correspond to palette registers (determined by the workspace base mode) and sound registers (not part of the display system).

All values are words in the workspace list; its format is:

Offset	**Value**
0	0 (indicates format of list)
4	Workspace base mode
8	Mode variable index
12	Mode variable value
16	Mode variable index
20	Mode variable value
...	...
n	-1

The workspace base mode is the number of an existing operating system screen mode which is used to determine the values of mode variables not explicitly mentioned in the list. The mode variable indices are the same as for SWI OS_ReadModeVariable.

Note: for the palette to be set properly, a workspace base mode should be chosen which has the appropriate palette.

When the service is received, the module should check that R2 contains a mode that it knows about and that R3 holds a monitor type that is suitable for that mode. If not, the service should be passed on. If R3 holds -1 then the MOS is making a general enquiry about that mode (eg to determine the attributes of a sprite defined in that mode) so the module should only check R2.

Note that it is possible for a mode to have two or more different sets of VIDC parameters for different monitor types, but the workspace parameters **must** be the same, as the mode number is used as an identifier in sprites and in calls such as OS_ReadModeVariable.

Service_ModeTranslation (Service Call &51)

Translate modes for unknown monitor types

On entry

R1 = &51 (reason code)
R2 = mode number that requires translation
R3 = monitor type

On exit

All registers preserved (if not claimed)

If claimed:
R1 = 0
R2 = substitute mode
R3 preserved

Use

This service is offered during a call to OS_CheckModeValid or a screen mode change, if the selected mode is not available with the current monitor type (this having been ascertained by offering Service_ModeExtension) and the monitor type is not one known to the MOS (ie not in the range 0 - 3 for RISC OS 2.0, or 0 - 5 for RISC OS 2.5).

If the monitor type passed in R3 is known to the module, then the module should discover what the attributes of the mode in R2 are (by calling OS_ReadModeVariable) and then choose a mode which is suitable for this monitor type and is closest in attributes to the selected mode. This mode number should be returned in R2.

Service_MonitorLeadTranslation (Service Call &76)

Translate monitor lead ID

On entry

R1 = &76 (reason code)
R2 = monitor lead ID (see below)

On exit

If monitor lead ID is recognised, then the module should set:

R1 = 0 (claim service)
R3 = default screen mode number to use on this type of monitor
R4 = monitor type number to use (as used in *Configure MonitorType)
R5 = sync type to use on this type of monitor
(0 ⇒ separate syncs, 1 ⇒ composite sync)

All other registers must be preserved.

If the monitor lead ID is not recognised, the module should preserve all registers.

Use

The monitor connector provides 4 ID pins, ID0-ID3. Each of these may be connected to 0v, +5v or to the Hsync pin. The monitor lead ID therefore represents the state of the 4 ID pins by 8 bits as follows:

Bit 0	Bit 1	State of ID0
Bit 2	Bit 3	State of ID1
Bit 4	Bit 5	State of ID2
Bit 6	Bit 7	State of ID3

0	0	Tied to 0v
1	0	Tied to +5v
0	1	Tied to Hsync
1	1	Indeterminate - either the state is fluctuating or machine is not capable of reading the ID

The service is issued when SWI OS_ReadSysInfo is called with R0=1 if any of the configured Mode, MonitorType or Sync are set to Auto.

If the service is not claimed, then RISC OS checks the monitor lead ID against the following list of recognised IDs:

Monitor ID pins Pin 0 1 2 3	Monitor type	Sync type	Default mode
1 1 H X	1 (Multisync)	1 (composite)	27
1 0 1 X	3 (Mono VGA)	0 (separate)	27
0 1 1 X	3 (Colour VGA)	0 (separate)	27
0 1 0 X	4 (Super VGA)	0 (separate)	27
H 1 1 X	0 (TV standard)	1 (composite)	12

0 = 0v
1 = +5v
H = Hsync
X = don't care

For all other ID values RISC OS uses the TV standard monitor settings.

SWI Calls

OS_Byte 9 (SWI &06)

Write duration of first flash colour

On entry

R0 = 9 (reason code)
R1 = new duration to write

On exit

R0 = preserved
R1 = duration before being overwritten
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the duration of the first flash colour.

Flashing colours are displayed as a sequence of two alternating colours. By default, each colour is displayed for 25 video frames at a time, which is approximately 0.5 seconds for 50Hz screen modes in the UK. This command allows you to alter the duration for which the first colour is displayed as follows:

Value	Meaning
0	Set an infinite duration (first colour constantly displayed)
n	Set the duration to n video frames (approximately n/50 seconds)

This variable may also be set using VDU 23,9. It may be read (but not set) by OS_Byte 195 (page 2-168).

Related SWIs

OS_Byte 10 (page 2-138), OS_Byte 195 (page 2-168)

Related vectors

ByteV

OS_Byte 10 (SWI &06)

Write duration of second flash colour

On entry

R0 = 10
R1 = duration to write

On exit

R0 preserved
R1 = duration before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call sets the duration for the second flash colour. See OS_Byte 9 for an explanation.

This variable may also be set using VDU 23,10. It may be read (but not set) by OS_Byte 194.

Related SWIs

OS_Byte 9 (SWI &06), OS_Byte 194 (SWI &06)

Related vectors

ByteV

OS_Byte 19 (SWI &06)

Wait for vertical sync

On entry

R0 = 19

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The video display frame is drawn approximately fifty times a second for most screen modes in the UK. This call synchronises a software routine with the signal produced when the video output reaches the bottom of the displayed area of the picture (ie the start of the border).

From this time until the next frame starts to be displayed (≈3.5ms for modes 0–17 and 24, ≈0.8ms for modes 18–21 and 23, and ≈1.4ms for modes 25-28), you have this time to redraw the screen.

It is possible to have more than this time by drawing from top to bottom, or setting a timer to wait until video output has passed the place on the screen you want to redraw.

If even this is not enough time to produce a flicker-free update of the screen, you should consider using more than one bank of screen memory and switching between them (using OS_Bytes 112–113 for example).

Related SWIs

OS_Byte 112 (SWI &06), OS_Byte 113 (SWI &06)

Related vectors

ByteV

OS_Byte 20 (SWI &06)

Reset font definitions

On entry

R0 = 20

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The shape of the character displayed when printing ASCII codes 32–255 may be redefined using the VDU 23,32–255 commands. Any such changes remain in force until the next hard reset. This command may be used to restore the default character definitions for ASCII codes in the range 32–127.

Note that you should really only redefine characters in the range 128–159. This is because all of the other printable characters have standard meanings which should be preserved for use in applications such as word processors.

See OS_Byte 25 for details on how to restore the other codes or how to restore a smaller selected group.

Related SWIs

OS_Byte 25 (SWI &06)

Related vectors

ByteV

OS_Byte 25 (SWI &06)

Reset group of font definitions

On entry

R0 = 25
R1 = group to restore

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

All ASCII characters between 32 and 255 may be redefined using the VDU 23 command. This call restores all or a particular group of characters to their default settings according to R1, as follows:

Value	**Range of characters to restore**
0	32–255
1	32–63
2	64–95
3	96–127
4	128–159
5	160–191
6	192–223
7	224–255

Related SWIs

OS_Byte 20 (SWI &06)

Related vectors

ByteV

OS_Byte 106 (SWI &06)

Select pointer/activate mouse

On entry

R0 = 106
R1 = pointer shape and linkage flag

On exit

R0 preserved
R1 = shape and linkage flag before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

You can define four 'pointer buffers' using OS_Word 21, each holding a different shape definition for the mouse pointer. This call allows you to select one of these definitions for future use, or to turn off the pointer depending on the bottom seven bits of R1:

Value	**Meaning**
0	Turn off current pointer
1–4	Select given pointer

If a pointer is selected it can be linked to the mouse so the mouse drives it, depending on bit seven of R1 as follows:

Value	**Meaning**
&00	Link pointer to mouse
&80	Pointer unlinked

For example, a value in R1 of &03 selects pointer three and links it to the mouse, and a value of &82 selects pointer two but leaves it unlinked.

Related SWIs

OS_Word 21 (SWI &07)

Related vectors

ByteV

OS_Byte 112 (SWI &06)

Write VDU driver screen bank

On entry

R0 = 112
R1 = bank number

On exit

R0 preserved
R1 = previous bank number
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call selects the bank of screen memory which is to be used by the VDU drivers according to R1, as follows:

Value	**Bank**
0	Default for the current screen mode (1 or 2)
n	Select bank 'n'

The maximum value for 'n' is (TotalScreenSize)/(ModeSize), where TotalScreenSize is the amount actually present in screen memory and ModeSize is the size of the current mode. For example, in mode 0, a 20K mode with 160K set aside for the screen makes eight banks available, so 8 is the maximum value for 'n'.

The default bank for a non-shadow mode is bank 1; for a shadow mode it is bank 2. OS_Byte 250 may be used to read the bank number without writing it

Related SWIs

OS_Byte 250 (SWI &06)

Related vectors

ByteV

OS_Byte 113 (SWI &06)

Write display hardware screen bank

On entry

R0 = 113
R1 = bank number

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call selects the bank of screen memory which is to be used by the display hardware according to R1:

Value	**Bank**
0	Default for the current screen mode
n	Select bank n

The bank may be read (but not set) using OS_Byte 251.

Related SWIs

OS_Byte 251 (SWI &06)

Related vectors

ByteV

OS_Byte 114 (SWI &06)

Write shadow/non-shadow state

On entry

R0 = 114
R1 = shadow state

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call determines whether future MODE commands will be forced into the shadow state, depending on R1:

Value	**Meaning**
0	Modes will be shadow
1	Modes will be non-shadow

Shadow state requires twice the amount of RAM than the equivalent non-shadow mode since two copies of the screen are stored in memory. OS_Bytes 112 and 113 control the use of the banks.

To select a shadow state temporarily when in non-shadow mode, you can use the MODE 128+n convention. Future MODE commands will not be influenced by this.

Related SWIs

OS_Byte 112 (SWI &06), OS_Byte 113 (SWI &06)

Related vectors

ByteV

OS_Byte 117 (SWI &06)

Read VDU status

On entry

R0 = 117

On exit

R0 preserved
R1 = status byte

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the content of the VDU status byte. This byte gives information on the way in which characters are output according to their bit settings:

Bit	Status when set
0	Printer output enabled by VDU 2
1	Unused
2	Paged scrolling selected by VDU 14
3	Text window in force (ie software scrolling)
4	In a shadow mode
5	In VDU 5 mode
6	Cursor editing in progress
7	Screen disabled with VDU 21

Related SWIs

None

Related vectors

ByteV

OS_Byte 134 (SWI &06)

Read text cursor position

On entry

R0 = 134

On exit

R0 preserved
R1 = position in x direction
R2 = position in y direction

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the current text cursor position unless cursor editing is in progress, in which case the position returned is that of the input cursor. OS_Byte 165 reads the position of the output cursor irrespective of cursor editing mode.

Text is printed at x positions 0 to n–1, where 'n' is the number of characters per line in the current text window. Therefore, the value obtained is normally in this range. However, if there is a pending newline (see VDU 23,16), a position of 'n' will be returned.

Related SWIs

OS_Byte 165 (SWI &06)

Related vectors

ByteV

OS_Byte 135 (SWI &06)

Read character at text cursor position and screen mode

On entry

R0 = 135

On exit

R0 preserved
R1 = ASCII value of character (0 if unreadable)
R2 = screen mode

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the screen mode and the ASCII code of the character at the text cursor position. If cursor editing is in progress, it returns the character code returned by the character at the input cursor position (ie the character that would be copied as input the next time Copy is pressed).

Note that the screen mode does not have bit 7 set, even if it is a shadow mode.

Related SWIs

None

Related vectors

ByteV

OS_Byte 144 (SWI &06)

Set vertical screen shift and interlace

On entry

R0 = 144
R1 = vertical screen shift (as a signed 8 bit number)
R2 = interlace flag

On exit

R0 preserved
R1 = previous vertical screen shift
R2 = previous interlace flag

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call specifies the vertical screen alignment and interlace options after the next mode change. R1 sets the vertical offset. R2 turns interlace on and off as follows:

Value	**Meaning**
0	Interlace on
1	Interlace off

This is equivalent to *TV, which is described in this chapter.

Related SWIs

None

Related vectors

ByteV

OS_Byte 160 (SWI &06)

Read VDU variable value

On entry

R0 = 160
R1 = VDU variable number (0–15)

On exit

R0 preserved
R1 = value of variable
R2 = value of next variable (R1 on entry + 1)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The VDU driver uses a number of locations in RAM to store transient information. This call allows some of these locations to be examined. Note that the variables are not necessarily stored in the order implied by the value of R1 on entry. However, the relationship between R1 and the variable read is guaranteed to remain the same for all versions of RISC OS.

Value	**Location**
0	LSB of graphics window left column (ic)
1	MSB of graphics window left column (ic)
2	LSB of graphics window bottom row (ic)
3	MSB of graphics window bottom row (ic)
4	LSB of graphics window right column (ic)
5	MSB of graphics window right column (ic)
6	LSB of graphics window top row (ic)
7	MSB of graphics window top row (ic)
8	Text window left column
9	Text window bottom row
10	Text window right column
11	Text window top row
12	LSB of graphics origin X coordinate (ec)
13	MSB of graphics origin X coordinate (ec)
14	LSB of graphics origin Y coordinate (ec)
15	MSB of graphics origin Y coordinate (ec)

- (ic) means internal coordinates: the origin is always the bottom left of the screen. One unit is one pixel wide and one pixel high.
- (ec) means external coordinates: a pixel is (1 « XEigFactor) units wide and (1 « YEigFactor) units high, where XEigFactor and YEigFactor are VDU variables.

This OS_Byte is provided mainly for compatibility with the BBC/Master 128. You can read many more of the VDU variables using OS_ReadVduVariables and OS_ReadModeVariable.

Related SWIs

OS_ReadVduVariables (SWI &31), OS_ReadModeVariable (SWI &35)

Related vectors

ByteV

OS_Byte 163 (SWI &06)

Read/write general graphics information

On entry

R0 = 163
R1 = 242
R2 = dot-dash repeat length or action code

On exit

R0 preserved
R1 = status, or preserved
R2 = status, or preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call is a general purpose one reserved for Acorn applications. The only value of R1 which is guaranteed to perform a useful function is 242. The type of action depends on the value of R2:

Value	Meaning
0	Set default dot-dash pattern and length
1–64	Set dot-dash line repeat length to the value given
65	Return status information
66	Return information on the current sprite

The status information is returned in R1 and R2 as follows:

R1 bits	Meaning
Bit 7 = 1	Sprites are always active
Bit 6 = 1	Flood fill is always active
Bits 0–5	Current dot dash line repeat length (0 means 64)

R2 bits	Meaning
Bits 0–31	Current size of the system sprite area in bytes.

The information on the current sprite is returned in R1 and R2 as follows:

R1 = width in pixels (ie internal coordinates)
R2 = height in pixels (ie internal coordinates)

Related SWIs

None

Related vectors

ByteV

OS_Byte 165 (SWI &06)

Read output cursor position

On entry

R0 = 165

On exit

R0 preserved
R1 = position in x direction
R2 = position in y direction

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the position of the output cursor, even while cursor editing is in progress.

Related SWIs

None

Related vectors

ByteV

OS_Byte 193 (SWI &06)

Read/write flash counter

On entry

R0 = 193
R1 = 0 to read or new duration to write
R2 = 255 to read or 0 to write

On exit

R preserved
R1 = duration before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The duration stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((duration AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call accesses the location used as a count-down timer for the flashing colours. The location is loaded with the count for the first colour and decremented at a VSync rate, providing that the current flash period is not infinite. When it reaches zero, the colours are swapped and the counter is loaded with the duration of the second colour.

Related SWIs

None

Related vectors

ByteV

OS_Byte 194 (SWI &06)

Read duration of second colour

On entry

R0 = 194
R1 = 0
R2 = 255

On exit

R0 preserved
R1 = duration
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This command will read the location that has been set by OS_Byte 10.

This must not be used to write this location, as RISC OS would then not match the location value. This call is only included for compatibility.

Related SWIs

OS_Byte 10 (SWI &06)

Related vectors

ByteV

OS_Byte 195 (SWI &06)

Read duration of first colour

On entry

R0 = 195
R1 = 0
R2 = 255

On exit

R0 preserved
R1 = duration
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This command will read the location that has been set by OS_Byte 9.

This must not be used to write this location, as RISC OS would then not match the location value. This call is only included for compatibility.

Related SWIs

OS_Byte 9 (SWI &06)

Related vectors

ByteV

OS_Byte 211 (SWI &06)

Read/write bell channel

On entry

R0 = 211
R1 = 0 to read or new channel to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = channel before being overwritten
R2 = bell sound information (see OS_Byte 212)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The channel stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((duration AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

The bell (VDU 7) sound is output on channel 1 by default. This call provides a means of determining the current channel or changing it if required.

Related SWIs

OS_Byte 212 (SWI &06), OS_Byte 213 (SWI &06), OS_Byte 214 (SWI &06)

Related vectors

ByteV

OS_Byte 212 (SWI &06)

Read/write bell volume

On entry

R0 = 212
R1 = 0 to read or new volume to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = volume before being overwritten
R2 = bell frequency (see OS_Byte 213)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The volume stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((duration AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This allows you to read or set the volume of the sound used to make the Ctrl-G bell sound. Values for the amplitude are in the range &80 (loudest) to &F8 (softest) in steps of &08. The default setting depends on the *Configure Loud/Quiet setting (&90/&D0 respectively).

Related SWIs

OS_Byte 211 (SWI &06), OS_Byte 213 (SWI &06), OS_Byte 214 (SWI &06)

Related vectors

ByteV

OS_Byte 213 (SWI &06)

Read/write bell frequency

On entry

R0 = 213
R1 = 0 to read or new frequency to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = frequency before being overwritten
R2 = bell duration (see OS_Byte 214)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The frequency stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((duration AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call provides a means of reading or changing the frequency associated with the bell sound. The default value is 100, and it has the same interpretation as the *Sound command.

Note that all frequencies are provided for compatibility only; new RISC OS values cannot be used.

Related SWIs

OS_Byte 211 (SWI &06), OS_Byte 212 (SWI &06), OS_Byte 214 (SWI &06)

Related vectors

ByteV

OS_Byte 214 (SWI &06)

Read/write bell duration

On entry

R0 = 214
R1 = 0 to read or new duration to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = duration before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The duration stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((duration AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call provides a means of reading or changing the duration of the bell sound. The default value is 6, and the unit is 20ths of a second.

Related SWIs

OS_Byte 211 (SWI &06), OS_Byte 212 (SWI &06), OS_Byte 213 (SWI &06)

Related vectors

ByteV

OS_Byte 217 (SWI &06)

Read/write paged mode line count

On entry

R0 = 217
R1 = 0 to read or new count to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = count before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The count stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((duration AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

In the paged output mode, the display is prevented from scrolling (awaiting the depression of Shift) when approximately 75% of the height of the current text window has been scrolled. The number of lines printed since the last page halt is maintained in the location accessed by this call and it may be either read or changed (normally to 0 before requesting user input).

If you are using OS_Word 0 or OS_ReadLine to perform the input, this call is made automatically. OS_Word 0 is provided for compatibility only and should not be used.

Related SWIs

None

Related vectors

ByteV

OS_Byte 218 (SWI &06)

Read/write bytes in VDU queue

On entry

R0 = 218
R1 = 0 to read or new count to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = count before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The count stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((duration AND R2) XOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call affects the count of the number of characters which remain to be passed to the VDU driver in order to complete the current VDU sequence. The value is (minus the number of bytes left), and is held in 2's complement notation (eg. &FF means one byte to go). The call may be used to read the value or to change it (normally to zero, which has the effect of abandoning an incomplete VDU command).

You can use this call when an escape condition is acknowledged. This prevents the first few characters of an error message from being 'swallowed' by an incomplete VDU sequence.

Related SWIs

None

Related vectors

ByteV

OS_Byte 250 (SWI &06)

Read VDU driver screen bank number

On entry

R0 = 250
R1 = 0
R2 = 255

On exit

R0 preserved
R1 = screen bank used by VDU drivers
R2 = display screen bank (see OS_Byte 251)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This command will read the location that has been set by OS_Byte 112.

This must not be used to write this location, as RISC OS would then not match the location value.

Related SWIs

OS_Byte 112 (SWI &06)

Related vectors

ByteV

OS_Byte 251 (SWI &06)

Read display screen bank number

On entry

R0 = 251
R1 = 0
R2 = 255

On exit

R0 preserved
R1 = screen bank used by the display
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This command will read the location that has been set by OS_Byte 113.

This must not be used to write this location, as the hardware would then not match the location value.

Related SWIs

OS_Byte 113 (SWI &06)

Related vectors

ByteV

OS_Word 9 (SWI &07)

Read pixel logical colour

On entry

R0 = 9 (reason code)
R1 = pointer to parameter block
R1+0 = LSB of X coordinate
R1+1 = MSB of X coordinate
R1+2 = LSB of Y coordinate
R1+3 = MSB of Y coordinate

On exit

R0 preserved
R1 preserved:
R1+4 = the logical colour of the pixel specified.

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call determines the logical colour of the pixel at given coordinates on the graphics screen. If the colour is returned as &FF then either:

- the screen is in a 256 colour mode
- the pixel is off the screen
- the screen is in a non-graphics mode.

To overcome the ambiguity caused by 256 colour modes, you should use OS_ReadPoint (SWI &32) instead. This returns both the logical colour and tint. The OS_Word should be used for compatibility purposes only, since the tint is discarded.

Related SWIs

OS_ReadPoint (SWI &32)

Related vectors

WordV

OS_Word 10 (SWI &07)

Read a character definition

On entry

R0 = 10
R1 = pointer to parameter block
    R1+0 = ASCII code of character required

On exit

R0 preserved
R1 preserved:
    R1+1 = top row of definition
    ...
    R1+10 = bottom row of definition

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The characters displayed in all modes other than Teletext mode are defined as an eight-by-eight matrix of dots. This call enables you to read the definition for a specified ASCII code. However, the definitions returned for ASCII codes 0 to 31 and 127 (ie the non-printing characters) are not meaningful apart from the following characters:

Value	Information returned
2	ECF pattern 1 (in native mode)
3	ECF pattern 2 (in native mode)
4	ECF pattern 3 (in native mode)
5	ECF pattern 4 (in native mode)
6	Dot-dash pattern

Bits set in each row of the character definition are displayed in the current text foreground colour; bits clear in each row are displayed in the current text background colour. In VDU 5 mode, bits which are set are plotted in the graphics foreground colour and action; bits which are clear are not plotted at all.

Related SWIs

None

Related vectors

WordV

OS_Word 11 (SWI &07)

Read the palette

On entry

R0 = 11
R1 = pointer to parameter block
  R1+0 = logical colour to read

On exit

R0 preserved
R1 preserved:
  R1+1 = physical colour associated with the specified logical colour
  R1+2 = red component
  R1+3 = green component
  R1+4 = blue component

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call allows you to determine the physical colour associated with a particular logical colour. The call can only return one of the colours associated with a flashing colour. To read the full information about a logical colour's palette entry, or to read the border and pointer palettes, you should use OS_ReadPalette (SWI &2F). The OS_Word is provided for compatibility only.

Related SWIs

OS_ReadPalette (SWI &2F)

Related vectors

WordV

OS_Word 12 (SWI &07)

Write the palette

On entry

R0 = 12
R1 = pointer to parameter block
R1+0 = logical colour to change
R1+1 = new physical colour
R1+2 = red component
R1+3 = green component
R1+4 = blue component

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call allows you to change the physical colour associated with a particular logical colour. It duplicates the function of VDU 19 command. However, the OS_Word call is faster and may be used in interrupt routines. The five bytes of the parameter block are equivalent to the five parameters l,p,r,g,b described in the section on VDU 19.

Related SWIs

None

Related vectors

WordV

OS_Word 13 (SWI &07)

Read current and previous graphics cursor positions

On entry

R0 = 13
R1 = pointer to parameter block

On exit

R0 preserved
R1 preserved:
R1+0 = LSB of previous X coordinate
R1+1 = MSB of previous X coordinate
R1+2 = LSB of previous Y coordinate
R1+3 = MSB of previous Y coordinate
R1+4 = LSB of current X coordinate
R1+5 = MSB of current X coordinate
R1+6 = LSB of current Y coordinate
R1+7 = MSB of current Y coordinate

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

All the coordinates are in external form. You can read points visited before the previous one (and many other VDU variables) using OS_ReadVduVariables (SWI &31).

Related SWIs

OS_ReadVduVariables (SWI &31)

Related vectors

WordV

OS_Word 21,0 (SWI &07)

Define pointer size, shape and active point

On entry

R0 = 21
R1 = pointer to parameter block

R1+0 =	0
R1+1 =	Shape number (1–4)
R1+2 =	Width (w) in bytes (0–8)
R1+3 =	Height (h) in pixels (0–32)
R1+4 =	ActiveX in pixels from left (0–(w*4–1))
R1+5 =	ActiveY in pixels from top (0–(h–1))
R1+6 =	Least significant byte of pointer (P) to data
R1+7	...
R1+8	...
R1+9 =	Most significant byte of pointer to data

On exit

R0, 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

You can define four shapes. These are numbered one to four and may be selected using OS_Byte 106.

As the pointer is always displayed in 2 bits per pixel (four pixels per byte), and the maximum width in bytes is 8, the maximum width is 32 pixels.

The ActiveX and ActiveY entries give the distance of the cursor 'hot spot' from the top left corner of the pointer. If these are zero, then positioning the pointer at coordinates (x, y) will move the top left corner to that position. Suppose the shape was a cross-hair 9 pixels in each direction; then making ActiveX and ActiveY (5,5) would position the hot-spot at the centre of the cross.

The data for the shape is pointed to by R1+6–R1+9. This data table contains the information for each row, from top to bottom, and the data within each row is given from left to right. Each byte contains the colours for four pixels. Bits 0,1 hold the colour number for the left-most pixel, bits 6,7 the colour for the right-most pixel. (So the pixels are displayed in reverse order to the order in which the byte would be written down.)

Colour zero is always transparent (ie the screen information shows through pixels in this colour). The other three colours may be set independently of any other colours on the screen using VDU 19 or the equivalent OS_Word.

However, note that colour two should be used with caution in defining pointer shapes, as it does not work correctly on high-resolution mono screens.

Related SWIs

OS_Byte 106 (SWI &06)

Related vectors

WordV

OS_Word 21,1 (SWI &07)

Define mouse coordinate bounding box

On entry

R0 = 21
R1 = pointer to parameter block

R1+0 = 1 (sub-reason code)	
R1+1 = LSB of left coordinate	All treated as signed 16-bit values, relative to screen origin at the time the command is issued
R1+2 = MSB of left coordinate	
R1+3 = LSB of bottom coordinate	
R1+4 = MSB of bottom coordinate	
R1+5 = LSB of right coordinate	
R1+6 = MSB of right coordinate	
R1+7 = LSB of top coordinate	
R1+8 = MSB of top coordinate	

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The coordinates should be given as signed 16-bit values relative to the graphics origin at the time the command is issued.

If (left > right) or (bottom > top) then the command is ignored.

An infinite box can be obtained by setting:

left	=	&8000	(-32768)
bottom	=	&8000	(-32768)
right	=	&7FFF	(32767)
top	=	&7FFF	(32767)

If the current mouse position is outside the box, it is homed to the nearest point inside the box. The buffer is not flushed, but any buffered coordinates will be moved inside the bounding box when they are read

When the mode changes, the box is set to the size of the screen.

Related SWIs

None

Related vectors

WordV

OS_Word 21,2 (SWI &07)

Define mouse multipliers

On entry

R0 = 21
R1 = pointer to parameter block
R1+0 = 2
R1+1 = X multiplier (these are both treated as signed 8-bit values)
R1+2 = Y multiplier

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The multipliers control the ratio between the movement of the mouse and the change in the coordinates of the mouse. The higher each value, the greater the amount the pointer moves (if linked to the mouse) for a given movement of the mouse.

The multipliers should both be given as signed eight-bit values. By specifying negative values (eg. 255 for -1), you can make the point move in the opposite direction from usual.

Both multipliers default to the configured MouseStep value. This is initially 1, when a movement of approximately 15cm of the mouse will move the pointer across the screen.

Related SWIs

None

Related vectors

WordV

OS_Word 21,3 (SWI &07)

Set mouse position

On entry

R0 = (reason code)
R1 = pointer to parameter block
R1+0 = 3
R1+1 = LSB of X position
R1+2 = MSB of X position
R1+3 = LSB of Y position
R1+4 = MSB of Y position

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The new values for the X and Y positions of the mouse are given as two signed 16-bit values. If the new position lies outside the bounding box of the mouse, this command will be ignored.

Note that this call sets the position of the mouse rather than the pointer. If the mouse and pointer are not linked, the position of the pointer on the screen is left unchanged.

Related SWIs

None

Related vectors

WordV

OS_Word 21,4 (SWI &07)

Read unbuffered mouse position

On entry

R0 = 21
R1 = pointer to parameter block
R1+0 = 4

On exit

R0 preserved
R1 preserved:

R1+1	= LSB of X position
R1+2	= MSB of X position
R1+3	= LSB of Y position
R1+4	= MSB of Y position

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call will read the position of the mouse at the time of the call. That is, it will not read the position from the mouse buffer.

Care should be taken when reading this position, as the buffer positions may be significantly out of step.

With RISC OS 2.0 this call generates an undefined instruction trap due to a stack mismatch. This has been fixed in RISC OS 2.5.

Related SWIs

None

Related vectors

WordV

OS_Word 21,5 (SWI &07)

Set pointer position

On entry

R0 = 21 (reason code)
R1 = pointer to parameter block
R1+0 = 5
R1+1 = LSB of X position
R1+2 = MSB of X position
R1+3 = LSB of Y position
R1+4 = MSB of Y position

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The new values for the X and Y positions of the pointer are given as two signed 16-bit values.

Note that this call sets the position of the pointer rather than the mouse. If the mouse and pointer are linked, then the pointer position will be updated with the mouse position on the next VSync interrupt.

Related SWIs

None

Related vectors

WordV

OS_Word 21,6 (SWI &07)

Read pointer position

On entry

R0 = 21
R1 = pointer to parameter block
  R1+0 = 6

On exit

R0 preserved
R1 preserved:
  R1+1 = LSB of X position
  R1+2 = MSB of X position
  R1+3 = LSB of Y position
  R1+4 = MSB of Y position

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call will read the position of the pointer. If the mouse and pointer are not linked, then this call read the position that the pointer was last set to.

If they are linked, then the pointer is updated from the unbuffered mouse position every VSync; otherwise 9 clicks while the Desktop is busy would freeze the pointer.

Related SWIs

None

Related vectors

WordV

OS_Word 22 (SWI &07)

Write screen base address

On entry

R0 = 22
R1 = pointer to parameter block
R1+0 = Type
R1+1 = Least significant byte of offset
R1+2...
R1+3...
R1+4 = Most significant byte of offset

On exit

R0, R1 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This routine sets up a new screen base address. It is given as the offset from the address of the base of the screen buffer to the start of the screen display. This address can be used as the area of the buffer which is to be updated, ie written to by the VDU drivers, or the area which is to be displayed by the hardware, or both, depending on the bits of the first byte in the parameter block:

Bit 0	Used by VDU drivers
Bit 1	Displayed by hardware

This allows multiple screens to be used. For example, in mode 12 two copies of the screen can be kept. One of these can be updated whilst the other is being displayed using the following parameter blocks:

R1+0	Contains 2	Displayed
R1+1–R1+4	Contains &00	
R1+0	Contains 1	Updated
R1+1–R1+4	Contains &14000	

Then the two screens can be swapped over (at VSync) by changing over the addresses so that smooth animation is obtained.

The configured ScreenSize determines the amount of RAM initially set aside for the screen. This can subsequently change, for example if you drag the **screen memory** bar in the Task Manager, or call OS_ChangeDynamicArea. You can read the current amount set aside for the screen by reading the VDU variable TotalScreenSize; and you can read the amount needed for a single screen by reading the mode variable ScreenSize.

A slightly simpler way of achieving bank switching is to use OS_Bytes 112–113. With these, you only have to specify the bank number, not the actual offset.

Related SWIs

OS_Byte 112 (SWI &06), OS_Byte 113 (SWI &06)

Related vectors

WordV

OS_Mouse (SWI &1C)

Read a mouse state from the buffer

On entry

–

On exit

R0 = mouse X coordinate
R1 = mouse Y coordinate
R2 = mouse buttons
R3 = time of button change

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered as interrupts are disabled

Use

OS_Mouse reads from the mouse buffer the mouse X and Y positions as values between –32768 and 32767. Unless the graphics origin has been changed, the coordinates will lie within the mouse bounding box, which initially defaults to the screen area. The call also returns buttons currently pressed as a value in the range 0–7:

Bit	Meaning when set
0	Right button down
1	Middle button down
2	Left button down

If there is no entry in the mouse buffer, the current status is returned. R3 gives the time the entry was buffered, or the current time if it is not a buffered reading. It uses the monotonic timer (see OS_ReadMonotonicTime).

Related SWIs

OS_ReadMonotonicTime (SWI &42)

Related vectors

MouseV

OS_ReadPalette (SWI &2F)

Read the palette setting of a colour

On entry

R0 = logical colour
R1 = type of colour

On exit

R2 = setting of first flashing colour
R3 = setting of second flashing colour

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_ReadPalette reads the setting of a particular colour that is sent to the hardware. R1 selects whether the normal colour, border colour or pointer colour is read as follows:

Value	Meaning
16	Read normal colour
24	Read border colour
25	Read pointer colour

The settings for the first flash colour and second flash colour are returned in R2 and R3 respectively. If these are identical then the colour is a steady, non-flashing one. The value contained in each of these is interpreted as follows:

Bits	Meaning
0–6	Value showing how colour was programmed
7	Supremacy bit
8–15	Amount of red
16–23	Amount of green
24–31	Amount of blue

The bottom byte (bits 0–7) returns the value of the second parameter to the VDU 19 command which defines the palette (bit 7 is the supremacy bit). For example:

Value	Meaning
0–15	Actual colour (BBC compatible)
16	Defined by giving amounts of red, green and blue
17–18	Flashing colour defined by giving amounts of red, green and blue

Related SWIs

None

Related vectors

None

OS_ReadVduVariables (SWI &31)

Read a series of VDU variables

On entry

R0 = pointer to input block
R1 = pointer to output block

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_ReadVduVariables reads in a series of VDU variables and places them in sequence into a block of memory. The input block consists of a sequence of words. Each word is the number of the variable to be read. A value of –1 terminates the list. The value of each variable is put as a word into the output block, any invalid variables being entered as zero. The output block has no terminator. Both blocks must be word-aligned.

The possible variable numbers are the same as for OS_ReadModeVariable (see below) with the following additions:

Name	No.	Meaning
VIDCClockSpeed	127	VIDC clock rate in kHz (not available in RISC OS 2.0)
GWLCol	128	Left-hand column of the graphics window (ic)
GWBRow	129	Bottom row of the graphics window (ic)
GWRCol	130	Right-hand column of the graphics window (ic)
GWTRow	131	Top row of the graphics window (ic)

TWLCol	132	Left-hand column of the text window
TWBRow	133	Bottom row of the text window
TWRCol	134	Right-hand column of the text window
TWTRow	135	Top row of the text window
OrgX	136	X coordinate of the graphics origin (ec)
OrgY	137	Y coordinate of the graphics origin (ec)
GCsX	138	X coordinate of the graphics cursor (ec)
GCsY	139	Y coordinate of the graphics cursor (ec)
OlderCsX	140	X coordinate of oldest graphics cursor (ic)
OlderCsY	141	Y coordinate of oldest graphics cursor (ic)
OldCsX	142	X coordinate of previous graphics cursor (ic)
OldCsY	143	Y coordinate of previous graphics cursor (ic)
GCsIX	144	X coordinate of graphics cursor (ic)
GCsIY	145	Y coordinate of graphics cursor (ic)
NewPtX	146	X coordinate of new point (ic)
NewPtY	147	Y coordinate of new point (ic)
ScreenStart	148	Address of the start of screen used by VDU drivers
DisplayStart	149	Address of the start of screen used by display hardware
TotalScreenSize	150	Amount of memory currently allocated to the screen
GPLFMD	151	GCOL action for foreground colour
GPLBMD	152	GCOL action for background colour
GFCOL	153	Graphics foreground colour
GBCOL	154	Graphics background colour
TForeCol	155	Text foreground colour
TBackCol	156	Text background colour
GFTint	157	Tint for graphics foreground colour
GBTint	158	Tint for graphics background colour
TFTint	159	Tint for text foreground colour
TBTint	160	Tint for text background colour
MaxMode	161	Highest mode number available
GCharSizeX	162	X size of VDU 5 chars (in pixels)
GCharSizeY	163	Y size of VDU 5 chars (in pixels)
GCharSpaceX	164	X spacing of VDU 5 chars (in pixels)
GCharSpaceY	165	Y spacing of VDU 5 chars (in pixels)
HLineAddr	166	Address of fast line-draw routine
TCharSizeX	167	X size of VDU 4 chars (in pixels)
TCharSizeY	168	Y size of VDU 4 chars (in pixels)
TCharSpaceX	169	X spacing of VDU 4 chars (in pixels)
TCharSpaceY	170	Y spacing of VDU 4 chars (in pixels)
GcolOraEorAddr	171	Address of colour blocks for current GCOLs
VIDCClockSpeed	172	VIDC clock speed in kHz (eg 24000 ⇒ 24 MHz)

WindowWidth	256	Characters that will fit on a row of the text window without a newline being generated
WindowHeight	257	Rows that will fit in the text window without scrolling it

- *ic* means internal coordinates, where (0,0) is always the bottom left of the screen. One unit is one pixel.
- *ec* means external coordinates, where (0,0) means the graphics origin, and the size of one unit depends on the resolution. The number of external units on a screen is dependent upon the video mode used; for example MODE 16 has 1280 by 1024 external units. The graphics origin is stored in external coordinate units, but is relative to the bottom left of the screen.
- *new point* is the internal form of the coordinates given in an unrecognised PLOT command. When the UKPlot vector is called, the internal format coordinates (variables 140–145) have not yet been shuffled down, so the graphics cursor (144–5) contains the coordinates of the last point visited. The external coordinates version of the current point (138–9) is updated from the coordinate given in the unrecognised plot.
- HL*ineAddr* points to a fast horizontal line draw routine. It is called as follows:

  R0 = left x-coordinate of end of line

  R1 = y-coordinate of line

  R2 = right x-coordinate of end of line

R3 =	0	plot with no action (ie do nothing)
	1	plot using foreground colour and action
	2	invert current screen colour
	3	plot using background colour and action
	≥ 4	pointer to colour block (on 64-byte boundary):

Offset	**Value**
0	OR mask for top ECF line
4	exclusive OR mask for top ECF line
8	OR mask for next ECF line
12	exclusive OR mask for next ECF line
...	
56	OR mask for bottom ECF line
60	exclusive OR mask for bottom ECF line

  R14 = return address

  Must be entered in SVC mode

  All registers are preserved on exit

  All coordinates are in terms of pixels from the bottom left of the screen. The line is clipped to the graphics window, and is plotted using the colour action specified by R3. The caller must have previously called OS_RemoveCursors and call OS_RestoreCursors afterwards.

- GcolOraEorAddr points to colour blocks for current GCOLs. If the value returned is n, then:

  n+&00–n+&3F is a colour block for the foreground colour + action
  n+&40–n+&7F is a colour block for the background colour + action
  n+&80–n+&BF is a colour block for the background colour with store action

  Each colour block is as described above. These are updated whenever a GCOL or TINT is issued or the ECF origin is changed. They are intended for programs which want to access screen memory directly and have access to the current colour/action settings.

Related SWIs

OS_ReadModeVariable (SWI &35)

Related vectors

None

OS_ReadPoint (SWI &32)

Read the colour of a point

On entry

R0 = X coordinate
R1 = Y coordinate

On exit

R0, R1 preserved
R2 = colour
R3 = tint
R4 = screen flag

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The coordinates passed are in external units and are relative to the current graphics origin.

OS_ReadPoint takes a point and returns its colour in R2 and its tint setting (amount of white, in the range 0-255) in R3. R4 returns the following:

Value	**Meaning**
0	Point on the screen
-1	Point off the screen (R2 = -1 also)

See VDU 19 for a description of colour and tint values.

Related SWIs

None

Related vectors

None

OS_ReadModeVariable (SWI &35)

Read information about a screen mode

On entry

R0 = screen mode, or –1 for current mode
R1 = variable number

On exit

R0, R1 preserved
R2 = value of variable
the C flag is set if variable or mode numbers were invalid

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_ReadModeVariable allows you to read information about a particular screen mode without having to change into that mode. The possible variable numbers are given below:

Name	**No.**	**Meaning**		
ModeFlags	0	The bits of the result have the following meanings:		
		Bit 0	= 0	graphics mode
			= 1	non-graphics mode
		Bit 1	= 0	non-Teletext mode
			= 1	Teletext mode
		Bit 2	= 0	non-gap mode
			= 1	gap mode
		Bit 3	= 0	non-gap mode

		= 1	'BBC' gap mode (eg modes 3 and 6)
	Bit 4	= 0	not Hi-resolution mono mode
		= 1	Hi-resolution mono mode
	Bit 5	= 0	VDU characters are normal height
		= 1	VDU characters are double height
	Bit 6	= 0	hardware scroll used
		= 1	hardware scroll never used

ScrRCol	1	Maximum column number for printing text ie number of columns–1
ScrBRow	2	Maximum row number for printing text ie number of rows–1
NColour	3	Maximum logical colour ie either 1, 3, 15 or 63 (not 255)
XEigFactor	4	This indicates the number of bits by which an X-coordinate must be shifted right to convert to screen pixels. Thus if this value is n, then one screen pixel corresponds to 2^n external coordinates in the X-direction.
YEigFactor	5	This indicates the number of bits by which a Y-coordinate must be shifted right to convert to screen pixels. Thus if this value is n, then one screen pixel corresponds to 2^n external coordinates in the Y-direction.
LineLength	6	Number of bytes on a pixel row This is the same as (characters per row) * (bits per pixel) * (pixel width of character) / 8. For example, in mode 15 it is 80*8*8/8, or 640.
ScreenSize	7	Number of bytes one screen buffer occupies. This must be a multiple of 256 bytes.
YShftFactor	8	Scaling factor for start address of a screen row. This variable is kept for compatibility reasons and should not be used.
Log2BPP	9	LOG base 2 of the number of bits per pixel
Log2BPC	10	LOG base 2 of the number of bytes per character. It is in fact the LOG base 2 of the number of bytes per character divided by eight. So in mode 0, for example, it is LOG base 2 of (8/8), or 0. In mode 15 it is LOG base 2 of (64/8), or 3. It would be exactly the same as Log2BPP, except for the 'double pixel' modes.
XWindLimit	11	Number of X pixels on screen–1

YWindLimit	12	Number of Y pixels on screen–1

Related SWIs

OS_ReadVduVariables (SWI &31)

Related vectors

None

OS_RemoveCursors (SWI &36)

Remove the cursors from the screen

On entry

–

On exit

–

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_RemoveCursors removes the cursors (output and copy, if active) from the screen, saving the old state (their positions, flash rate etc.) on an internal stack so that it may be recovered later. This instruction must always be balanced later by a OS_RestoreCursors to restore the cursor again.

This call is provided only for routines that need direct screen access.

Note that routines that directly access the screen may need to run in SVC mode if the routines are to work with hardware scrolled screens, which may straddle the logical-physical memory boundary at 32MByte. If the routines do not need to work with hardware scrolled screens, then USR mode is adequate.

Related SWIs

OS_RestoreCursors (SWI &37)

Related vectors

None

OS_RestoreCursors (SWI &37)

Restore the cursors to the screen

On entry

–

On exit

–

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_RestoreCursors restores the cursor state previously saved on the internal stack using OS_RemoveCursors.

This call is provided only for routines that need direct screen access.

Related SWIs

OS_RemoveCursors (SWI &36)

Related vectors

None

OS_CheckModeValid (SWI &3F)

Check if it is possible to change to a specified mode

On entry

R0 = mode number to check

On exit

if C flag = 0 then mode is valid:
    R0 = preserved
if C flag = 1 then mode is invalid:
    R0 = –1 if mode is non-existent
    R0 = –2 if not enough memory
    R1 = mode that would be used
    R1 = –2 if unable to select alternative mode

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_CheckModeValid determines whether you can change to a given mode and return with the carry bit appropriately set. If the mode you are checking isn't available on the current type of monitor, then R1 will contain the mode that will be used if an attempt is made to select the mode which you are checking, using VDU 22. If there is insufficient memory or the call is unable to determine an alternative for another reason, then –2 will be returned.

If this call returns that there is insufficient memory for the required mode, then it can be borrowed from other areas of the machine. See the chapter entitled *Memory Management* on page 1-329 for details.

Related SWIs

None

Related vectors

None

OS_Plot (SWI &45)

Direct VDU call

On entry

R0 = plot command code
R1 = x coordinate
R2 = y coordinate

On exit

R0 - R2 corrupted

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is equivalent to a VDU 25 command. However, it is much more efficient as only one call is required (instead of six calls to OS_WriteC). The call goes directly to the VDU drivers unless spooling has been turned on, redirection has been turned on or if WrchV has been claimed.

Related SWIs

None

Related vectors

WrchV

OS_SetECFOrigin (SWI &56)

Set the origin of the ECF patterns

On entry

R0 = x coordinate
R1 = y coordinate

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

By default, the alignment of ECF patterns is with the bottom left corner of the screen. This command makes the bottom left of the pattern coincide with the bottom left of the specified point.

The origin is restored to the default after a mode change.

VDU 23,17,6 performs the same action.

Related SWIs

None

Related vectors

None

OS_ReadSysInfo (SWI &58)

Read screen size after hard reset

On entry

R0 = 0

On exit

R0 = size in bytes

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call will return the screen size in bytes after the next hard reset.

Related SWIs

None

Related vectors

None

OS_ChangedBox (SWI &5A)

Determine which area of the screen has changed

On entry

R0 =	0	disable changed box calculations
	1	enable changed box calculations
	2	reset changed box to null rectangle
	–1	read changed box information

On exit

R0 = previous enable state in bit 0 (0 for disabled, 1 for enabled)
R1 = pointer to a fixed block of 5 words, containing the following info:
R1+0 = new disable/enable flag (in bit 0)
R1+4 = x-coordinate of left edge of box
R1+8 = y-coordinate of bottom edge of box
R1+12 = x-coordinate of right edge of box
R1+16 = y-coordinate of top edge of box

The (R1+4) to (R1+16) values are only valid if the change box calculations were in an enabled state immediately after the call; otherwise they are undefined.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call checks which areas of the screen have changed over calls to the VDU drivers. When this feature is enabled, RISC OS maintains the coordinates of a rectangle which completely encloses any areas that have changed since the last time the rectangle was reset.

This is particularly useful for applications which switch output to sprites, and then want to repaint the sprite onto the screen after performing VDU operations on the sprite. The application can make significant speed improvements by only repainting the section of the sprite which corresponds to the changed box.

All coordinates are measured in pixels from the bottom left of the screen. If a module provides extensions to the VDU drivers, it should read the address of this block on initialisation, and update the coordinates as appropriate. If an exact calculation of which areas have been modified is difficult, then the module should extend the rectangle to include the whole of the graphics window (or indeed the whole screen, if the operation can affect areas outside the graphics window).

The disable/enable flag at offset 0 in the block is for information only – it must not be modified directly, as RISC OS holds the master copy of this flag.

Changed box calculations are disabled on a mode change. However, the disable/enable state and the coordinates of the rectangle form part of the information held in save areas when output is switched between the screen and sprites.

Related SWIs

None

Related vectors

None

*Commands

*Configure Loud

Sets the configured volume for the beep to its loudest volume.

Syntax

```
*Configure Loud
```

Parameters

None

Use

*Configure Loud sets the configured volume for the beep to its loudest volume.

The change takes effect on the next reset.

Related commands

*Configure Quiet

Related SWIs

OS_Byte 212 (SWI &06)

Related vectors

None

*Configure Mode

Sets the configured screen mode used

Syntax

```
*Configure Mode screen_mode
```

Parameter

screen_mode	the display mode that the computer should use after a power-on or hard reset, and when entering or leaving the desktop

Use

*Configure Mode sets the configured screen mode used by the machine when it is first switched on, or after a hard reset, and when entering or leaving the desktop. It is identical to the command *Configure WimpMode; the two commands alter the same value in CMOS RAM.

Under RISC OS 2.0, this command only sets the configured screen mode used for the command line; *Configure WimpMode sets the configured screen mode used for the Desktop.

Example

```
*Configure Mode 27
```
selects VGA mode with 16 colours

Related commands

*Configure WimpMode, VDU 22

Related SWIs

None

Related vectors

None

*Configure MonitorType

Sets the configured monitor type

Syntax

```
*Configure MonitorType n|Auto
```

Parameters

n	0 to 5

Use

*Configure MonitorType sets the configured monitor type that is connected to the computer. The values of n correspond to the following monitors:

n	**Monitor**	
0	50Hz TV standard colour or monochrome monitor	
1	Multiscan monitor	
2	Hi-resolution 64Hz monochrome monitor	
3	VGA-type monitor	
4	Super-VGA-type monitor	(not available in RISC OS 2.0)
5	LCD (liquid crystal display)	(not available in RISC OS 2.0)

You can also set a value of Auto (not available in RISC OS 2.0). More recent Acorn computers can sense the type of monitor lead connected, and hence set the monitor type. If no lead can be sensed, either because none is present or because the computer is of an older design, the monitor type defaults to 0.

You can also configure the monitor type by holding down the corresponding key from the numeric keypad while the computer is switched on.

Example

```
*Configure MonitorType 3
```
configure VGA-type monitor

Related commands

VDU 22

Related SWIs

None

Related vectors

None

*Configure MouseStep

Sets the configured value for how fast the pointer moves as you move the mouse

Syntax

```
*Configure MouseStep n
```

Parameters

n	a number between 1 and 127

Use

*Configure MouseStep sets the configured value for how fast the pointer moves as you move the mouse. Useful values of n are 1, 2 or 3 for slow, medium or fast, respectively.

The mouse position is moved by n coordinates for each movement of the mouse. Although values up to 127 are accepted, anything above 6 is impractical because the step is too large.

You can also use OS_Word 21,2 to set dynamically the mouse step.

Example

```
*Configure MouseStep 3                select a fast speed
```

Related commands

None

Related SWIs

OS_Word 21 (SWI &07)

Related vectors

None

*Configure NoScroll

Sets the configured scrolling so the screen does not scroll upwards at the end of a line

Syntax

```
*Configure NoScroll
```

Parameters

None

Use

*Configure NoScroll prevents a newline from being generated when a character is printed at the end of a line. The default value is Scroll.

When printing a character in VDU 4 mode results in the cursor moving beyond the edge of the window, a 'pending newline' is generated. It is actually executed just before the next character is printed, provided that it has not been deleted or executed by another cursor control character. For example VDU 127 would cancel it; VDU 9 would execute it.

Refer to VDU 23,16 for a lengthier description of NoScroll, and for details of how to set dynamically this option.

Related commands

*Configure Scroll

Related SWIs

None

Related vectors

None

*Configure Quiet

Sets the configured volume for the beep to half its loudest volume.

Syntax

```
*Configure Quiet
```

Parameters

None

Use

*Configure Quiet sets the configured volume for the beep to half its loudest volume.

The change takes effect on the next reset.

Related commands

*Configure Loud

Related SWIs

OS_Byte 212 (SWI &06)

Related vectors

None

*Configure ScreenSize

Sets the configured amount of memory reserved for screen display

Syntax

```
*Configure ScreenSize mK|n
```

Parameters

*m*K	number of kilobytes of memory reserved
n	number of pages of memory reserved; n ≤ 127

Use

*Configure ScreenSize sets the configured amount of memory reserved for screen display. The default value is 80Kbytes on a 0.5Mbyte machine, and 160Kbytes for all other machines.

You can also use OS_ChangeDynamicArea (SWI &2A) to alter dynamically the screen memory allocation. For more information, refer to the chapter entitled *Memory Management*.

You should not configure more than 480Kbytes of screen memory, due to limitations in the MEMC1 and MEMC1a memory controllers.

Example

```
*Configure ScreenSize 160K        reserve 160Kbytes for screen display
```

Related commands

None

Related SWIs

OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

*Configure Scroll

Sets the configured scrolling so the screen scrolls upwards at the end of a line

Syntax

```
*Configure Scroll
```

Parameters

None

Use

*Configure Scroll generates a newline automatically whenever a character is printed at the end of a line. This is the default value.

When printing a character in VDU 4 mode results in the cursor moving beyond the edge of the window, the cursor is instead moved to the negative X edge of the window and one line in the positive Y direction.

Refer to VDU 23,16 for a lengthier description of Scroll, and for details of how to set dynamically this option.

Related commands

*Configure NoScroll

Related SWIs

None

Related vectors

None

*Configure Sync

Sets the configured type of synchronisation for vertical sync output

Syntax

```
*Configure Sync 0|1
```

Parameters

0	vertical sync
1	composite sync

Use

*Configure Sync selects vertical sync (parameter of 0) or composite sync (parameter of 1) on the vertical sync output of the video connector. For any monitor currently supplied for use with Acorn computers, you should not change the configured type from its default value of 1.

Example

```
*Configure Sync 1
```

Related commands

None

Related SWIs

None

Related vectors

None

*Configure TV

Sets the configured vertical screen alignment and screen interlace

Syntax

```
*Configure TV [vert_align[[,]interlace]]
```

Parameters

vert_align	adjusts the vertical screen alignment 0 to 3 lines up (values of 0-3 respectively), or 1 to 4 lines down (values of 255-252 respectively)
interlace	switches screen interlace on (with a value of 0), or off (with a value of 1)

Use

*Configure TV sets the configured vertical screen alignment and screen interlace. The default values are 0,1 (no vertical alignment offset and interlace off).

Example

`*Configure TV 0,1`	the default value

Related commands

*TV

Related SWIs

None

Related vectors

None

*Shadow

Sets which bank of screen memory is used on subsequent mode changes

Syntax

```
*Shadow [0|1]
```

Parameters

0 or 1 or nothing

Use

*Shadow sets which bank of screen memory is used on subsequent changes to the screen mode. It controls two banks of screen memory: the normal bank (bank 1, known as the *non-shadow* bank), and an alternate bank (bank 2, known as the *shadow* bank).

If you give either no parameter, or a parameter of 1, the shadow bank is used on the next mode change. If you give a parameter of 0, the non-shadow bank is used on the next mode change.

For the shadow bank to be used, there must be at least double the memory for the selected screen mode available in the screen area of memory. For example, to use shadow memory in screen mode 8 (a mode which requires 40Kbytes), at least 80Kbytes of screen memory must be available.

Example

```
*Shadow 1
```

Related commands

*Configure ScreenSize

Related SWIs

None

Related vectors

None

*TV

Adjusts the vertical screen alignment and screen interlace

Syntax

```
*TV [vert_align[[,]interlace]]
```

Parameters

vert_align	adjusts the vertical screen alignment 0 to 3 lines up (values of 0-3 respectively), or 1 to 4 lines down (values of 255-252 respectively)
interlace	switches screen interlace on (with a value of 0), or off (with a value of 1)

Use

*TV adjusts the vertical screen alignment and screen interlace.

The change takes effect on the next mode change.

Example

```
*TV 3,0                    move the picture up 3 lines, and turn interlace on
```

Related commands

*Configure TV

Related SWIs

None

Related vectors

None

Application Notes

Examples of ECF pattern use

This section gives some examples of how you might set ECF patterns using the VDU 23,2-5... commands.

In BBC/Master compatible mode

For example in modes with four bits per pixel, bits 7, 5, 3 and 1 of the n parameter control the logical colour of the left-hand pixel, and bits 6, 4, 2 and 0 control the right-hand pixel. To set the left pixel to colour 2 (green by default) and the right one to colour 7 (white), the colours are combined as follows:

Pixel 1 colour (left)	Green	2	0010
Pixel 2 colour (right)	White	7	0111

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Left pixel	0		0		1		0	
Right pixel		0		1		1		1
Result	0	0	0	1	1	1	0	1

Resulting value = &1D (29)

Whereas in modes with two bits per pixel the method is:

Pixel 1 colour (left)	Yellow	2	10
Pixel 2 colour	Red	1	01
Pixel 3 colour	White	3	11
Pixel 4 colour (right)	Yellow	2	10

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Pixel 1	1				0			
Pixel 2		0				1		
Pixel 3			1				1	
Pixel 4				1				0
Result	1	0	1	1	0	1	1	0

Resulting value = &B6 (182)

In RISC OS native mode

In RISC OS mode, for example, in modes with four bits per pixel, the colour of the left-hand pixel is formed from bits 3, 2, 1 and 0 of the n parameter, and the colour of the right-hand pixel comes from bits 7, 6, 5 and 4 of the parameter. So, if the pixels are to be logical colours 2 and 7 again, the colours are combined as follows:

Pixel 1 colour (left)	Green	2	0010
Pixel 2 colour (right)	White	7	0111

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Right pixel	0	1	1	1				
Left pixel					0	0	1	0
Result	0	1	1	1	0	0	1	0

Resulting value = &72 (114)

Notice that the pixel colours on the left, as displayed, are derived from the bits on the right, as written down, and vice versa.

In modes with two bits per pixel the method is:

Pixel 1 colour (left)	Yellow	2	10
Pixel 2 colour	Red	1	01
Pixel 3 colour	White	3	11
Pixel 4 colour (right)	Yellow	2	10

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Pixel 4	1	0						
Pixel 3			1	1				
Pixel 2					0	1		
Pixel 1							1	0
Result	1	0	1	1	0	1	1	0

Resulting value = &B6 (182)

Further examples of ECF patterns

Here are examples of how to produce a pattern of alternating red (colour 1) lines and white (colour 7) lines (with the default palette). Each of the VDU 23,2 or VDU 23,12 commands alters ECF pattern 1 to cause the same effect.

in a 2 colour mode (black and white only available):

```
VDU 23,12,1,1,0,0,1,1,0,0
VDU 23,2,&FF,0,&FF,0,&FF,0,&FF,0
VDU 23,17,4,1| has no effect
```

in a 4 colour mode:

```
VDU 23,12,1,1,3,3,1,1,3,3
VDU 23,2,&0F,&FF,&0F,&FF,&0F,&FF,&0F,&FF
after VDU 23,17,4,1|
VDU 23,2,&55,&FF,&55,&FF,&55,&FF,&55,&FF
```

in a 16 colour mode:

```
VDU 23,12,1,1,7,7,1,1,7,7
VDU 23,2,3,&3F,3,&3F,3,&3F,3,&3F
after VDU 23,17,4,1|
VDU 23,2,&11,&77,&11,&77,&11,&77,&11,&77
```

in a 256 colour mode:

```
VDU 23,12,&C3,&FF,&C3,&FF,&C3,&FF,&C3,&FF
VDU 23,2,&17,&FF,&17,&FF,&17,&FF,&17,&FF
VDU 23,17,4,1| has no effect
```

22 Sprites

Introduction

A sprite is an area of memory that can be treated like a small block of screen memory. It contains a graphic shape made up of an array of pixels.

A sprite has the following attributes:

- a name used to identify the sprite, up to 12 characters in length
- the number of the screen mode whose format the sprite imitates
- a height and a width
- optionally, a transparency mask
- optionally, a palette defining the colours used in the sprite

If the sprite has a transparency mask, you can cause certain pixels in the sprite not to be written to the existing screen display. By using this mask, you can effectively make a sprite any shape.

A sprite can be defined by grabbing some or all of the screen, or defining it a pixel at a time or by making the VDU plot operations go into a sprite instead of the screen memory.

Once defined, a sprite can be manipulated in many ways, such as having rows and columns inserted or deleted, flipping it about the x or y axis and changing the colour of particular pixels.

A sprite can be plotted onto the screen scaled to any size, and its colours can be altered using a lookup table.

Sprites are stored in sprite files, which may contain one or more sprites with different names.

Overview

Sprite memory areas

RISC OS can use sprites from the *system sprite area*, or from any number of *user sprite areas*.

System sprite area

The first is the system sprite area, which is defined by the kernel. Its size can be controlled by a slider in the task manager application on the desktop.

This area is public and can be accessed from any program or module, so is a convenient place to experiment using sprites. However, you should not use the system sprite area in commercial applications and should instead use a combination of the Wimp's common sprite pool and user sprite area as appropriate.

Note that the Sprite module * Commands only work with sprites in the system sprite area.

User sprite area

Alternatively, an application or a module may reserve its own space. This is private space, which can only be used by the application or module that reserved it. For example, the Wimp has a shared sprite pool, which is passed to OS_SpriteOp as a user area.

Unlike the system area, there can be several user areas which are referenced via pointers to the start of the areas. In user areas, as well as being able to refer to a sprite by name, you can also refer to it by address. This plainly will be much faster, since there is no overhead to search through the available names.

Memory operations

With the sprite module, it is possible to issue calls to:

- clear a sprite area
- check how large an area is and how many sprites are in it
- scan through the list of names of sprites in an area

File operations

Sprites can be loaded and saved to any valid pathname. The simplest way of doing this is to use the calls to load or save the current graphics window as a single sprite file.

For more sophisticated control, a sprite area (system or user) can be saved, or loaded. It is also possible to merge a sprite file with what is already in memory.

Sprite files can be edited by the Paint application.

Creating sprites

You can create a blank sprite of a specified height and width. Subsequently, individual pixels can be changed within it.

You also have various ways of grabbing some or all of the graphics window and putting it into a sprite.

The various sprite editing utilities all use one or other of these techniques.

Mask control

The mask can be enabled and disabled as required. Like a sprite, it can have individual pixels set or cleared. A sprite may have up to 256 colours (64 palette entries stored), depending on which mode it was created in; the mask pixels are either on (solid), in which case the pixel colour is used, or off (transparent), in which case it is not plotted.

VDU output to sprite

The other way of writing to a sprite or its mask is to redirect the VDU operations to a sprite. This means that the sprite rectangle is treated like a graphics window, putting data into the sprite in the same format as the screen memory.

Sprite manipulation

Once a sprite is in memory, it can be manipulated in a number of ways, for example you can:

- rename, copy, delete the sprite or append it to another sprite
- insert or delete rows and columns
- flip about the x or y axis
- change an individual pixel's colour.

Plotting a sprite

There are several ways of plotting a sprite into the screen memory. There is a SWI that will simply plot the sprite. You can also plot it using the mask if one is attached to it. The scale of the sprite can be changed to be any desired size. Thus, zooming into a sprite is made very easy.

The anti-aliasing technique used by the font manager with characters can be used here with sprites. A range of close colours are used to shade the sprite, which can be plotted with or without a mask, and scaled to any size.

Technical Details

Common parameters

Several kinds of parameters are used by many SWIs within the sprite module. Rather than repeating their definitions each time, they are described here.

Pointer to control block of sprite area and sprite pointer

Many of the sprite SWIs use a pointer to control block of sprite area parameter in R1 or that and a sprite pointer in R2. When either of these appear, then bits 8 and 9 in R0 control how these two registers are interpreted.

R0 bit 8 & 9 values	R1 effect	R2 effect
00 (+0)	not used (system sprite area used)	pointer to sprite name
01 (+256)	pointer to user sprite area	pointer to sprite name
10 (+512)	pointer to user sprite area	pointer to sprite
11 (+768) is invalid		

Note that the sprite names are null terminated.

For example OS_SpriteOp 256+33,CBlock,NamePtr will interpret CBlock as a pointer to the user sprite area and use NamePtr as a pointer to the name of the sprite to use within that area.

Using a pointer to a sprite in the user area (R0+512) is the quickest way of using sprites, because the string lookup doesn't need to be done.

Scale factors

The scale factor will change the size of a sprite. It is a pointer to a block of four words with the following elements:

Offset	Meaning
0	x multiplier
4	y multiplier
8	x divisor
12	y divisor

The size of the specified sprite on the screen when it has been plotted in pixels (**not** OS units), is multiplied by the magnitude and divided by the divisor, ie:

x pixel size = x start size (in pixels) × x magnitude ÷ x divisor
y pixel size = y start size (in pixels) × y magnitude ÷ y divisor

If the plot action is using an ECF pattern, then the pattern will not be scaled up with the sprite. This is so that the patterning will be correct when used with a large scale factor. See the section entitled ECF *patterns* on page 2-44 for a description of ECF patterns.

If the pointer is zero, then no scaling is performed; ie 1:1 scale.

Pixel translation table

This allows a logical colour to be substituted for each colour in the sprite. It is a pointer to a table of bytes. The number of bytes in the table depends on the number of colours in the mode in which the sprite was created.

A pixel of colour N in the sprite will be translated to the Nth entry in the pixel translation table. The first entry in the table is at offset 0 (ie the 0th colour). So colour 3 in a pixel will get the value 3 bytes into the table and use that as its logical colour.

If the pointer is zero, then the colours in the sprite will be used. However, if the destination bits per pixel is less than the source bits per pixel, you will get an error.

The Wimp uses a similar system to provide mode independence. See *Wimp_SetPalette* (SWI &400E4), *Wimp_ReadPalette* (SWI &400E5), and *Wimp_SetColour* (SWI &400E6) on page 4-255 of the chapter entitled *The Window Manager* for details.

The ColourTrans module provides facilities for translation table calculations. For more information refer to the chapter entitled *ColourTrans* on page 4-381.

Plot action

The plot action is the way in which pixels are plotted onto the screen. Some SWIs use the VDU 18 setting, and others can be passed the number directly. In either case, the format is the same, apart from bit 3 (&08)

Value	**Action**
0	Overwrite colour on screen
1	OR with colour on screen
2	AND with colour on screen
3	exclusive OR with colour on screen
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND with colour on screen with NOT of sprite pixel colour
7	OR with colour on screen with NOT of sprite pixel colour
&08	If set then use the mask, otherwise don't
&10	ECF pattern 1
&20	ECF pattern 2
&30	ECF pattern 3
&40	ECF pattern 4
&50	Giant ECF pattern (patterns 1 - 4 placed side by side)

Save area

When output is switched to a sprite or its mask, it is possible to save the VDU context in a save area. The save area passed is where the state that has just been entered will be saved if **another** redirection of VDU output is made.

The save area is a block of memory, the size of which is obtained from OS_SpriteOp 62. The contents cannot be directly manipulated, but this is a list of the things that it stores:

- ECF patterns, BBC/Native ECF flag, ECF origin
- Dotted line pattern and length, and current position in pattern
- Graphics foreground and background actions, colours and tints
- Text foreground and background colours and tints
- Graphics and text window definitions
- Graphics origin
- Graphics cursor and two previous positions
- Text and input cursor positions
- VDU status (VDU 2 state, page mode, windowing, shadowing, VDU 5 mode, cursor editing state and VDU disabled/enabled)
- VDU queue and queue pointer
- Character sizes and spacings
- Changed box coordinates and status
- WrCh destinations flag
- Spool handle

Mode variables are reconstituted from the sprite mode number or the display mode number as appropriate.

The kernel maintains a save area for the screen (ie the system save area with a value 1). Therefore, if you swap output to a sprite, perform some operations and swap back, it will not be necessary to allocate a save area.

A save area that has not yet been used must have a zero in the first word. Once it has been used, then this is set to a non-zero value, so that when it is next passed to OS_SpriteOp 60 or OS_SpriteOp 61 the graphics state will be restored from it, rather than being set to the default state.

The use of save areas allows the VDU 'context' to be switched between various destinations, so that each area has its own separate VDU state.

Here are a couple of examples highlighting the above points. The first example shows how to set-up a once-off drawing into a sprite:

```
SYS "OS_SpriteOp",256+60,myarea,mysprite$,0 TO r0,r1,r2,r3
REM we don't need a save area, because nobody can swap output away from
REM our sprite; and we won't want to restore the state we're in when
REM we've finished our work on the sprite.
.... do whatever graphics we want ....
SYS "OS_SpriteOp",r0,r1,r2,r3
REM whatever output state was in force on entry is now restored
```

The second example shows how to draw into a sprite, interact with the user, while maintaining ECF patterns etc:

```
 SYS "OS_SpriteOp",256+62,myarea,mysprite$ TO ,,,size
 DIM sarea size
 sarea!0=0 : REM mark as unset
 REPEAT
   SYS"OS_SpriteOp",256+60,myarea,mysprite$,sarea TO r0,r1,r2,r3
     .... work on the sprite
   SYS "OS_SpriteOp",r0,r1,r2,r3: REM return to previous output
   REM at this point, our save area has been filled with our state;
   REM the next time we switch output to our sprite the OS variables
   REM will therefore be reset from it.
     ... talk to the user ...
UNTIL bored
```

Memory operations

To initialise the system sprite area, you can call OS_SpriteOp 9 or *SNew. To change the system sprite area size, you can call OS_ChangeDynamicArea (SWI &2A); you can also change the configured size of this area (which is used on a hard reset) by calling *Configure SpriteSize.

In order to setup a user sprite area, you must first allocate space for it using the usual memory allocation calls. You must then set up the header for the area before you call OS_SpriteOp 9 to initialise it as a sprite area.

Reading a sprite area

To check the state of a sprite area, *SInfo or OS_SpriteOp 8 will tell you how large the area is, how much has been used and how many sprites are in it. *SInfo will, of course, only work with the system area.

Finding the names of sprites

*SList will list the names of all sprites in the system area. OS_SpriteOp 13 allows you to find the name of a sprite given its number in the list. You would call OS_SpriteOp 8 first to find out how many sprites there are and then use this call to get the names one at a time.

File operations

The simplest sprite file operations are screen save and load. The screen save will take the entire graphics window and convert it into a sprite file. *ScreenSave and OS_SpriteOp 2 will perform this operation. *ScreenLoad and OS_SpriteOp 3 will load it back again, aligned with the bottom left hand corner of the current graphics window.

There is also a set of operations based around loading and saving sprite areas to a file. *SLoad and OS_SpriteOp 10 will load a sprite file into an initialised sprite area and set up all the pointers within it. To save, *SSave and OS_SpriteOp 12 will create a sprite file and write all the sprites from the specified sprite area into it.

The sprite load operations will delete all sprites currently in memory. If you wish to keep them, then *SMerge and OS_SpriteOp 11 will merge the sprite file sprites with those in memory. Any name clashes will result in the file sprite replacing the memory one.

Creating a sprite

There are two main ways of creating a sprite. You can grab a piece of screen memory using OS_SpriteOp 14 or 16, or *SGet. Alternatively, you can create a blank sprite with OS_SpriteOp 15 to be subsequently filled in. With this blank sprite, you can alter individual pixels or you can direct VDU operations into it. These are discussed later.

Creating a mask

To create a mask, OS_SpriteOp 29 must be used. It will initialise all the pixels solid, so that all of the sprite is plotted. You must alter it afterwards to set the mask that you require.

Sprite manipulation

The contents of a sprite may be manipulated in many ways.

Copy, rename or delete

You can copy, rename or delete a sprite in the following ways:

- To make a copy of a sprite, OS_SpriteOp 27 or *SCopy can be used. They will return an error if the designated name already exists.
- To rename a sprite, OS_SpriteOp 26 or *SRename can be used. Again, the same error condition applies to existing destination names.
- To delete a sprite, its mask and palette, OS_SpriteOp 25 or *SDelete can be used You can delete the mask of a sprite only, by calling OS_SpriteOp 30. Free space is automatically reclaimed in the sprite area.

Insert and delete row or column

You can insert and delete rows and columns at any place you wish in the sprite. These are the operations that you need to do this:

- OS_SpriteOp 31 to insert a row
- OS_SpriteOp 32 to delete a row
- OS_SpriteOp 45 to insert a column
- OS_SpriteOp 46 to delete a column

Axis flipping

A sprite can be flipped about its x or y axis. Flipped about the x axis using OS_SpriteOp 33 or *SFlipX will make it appear upside down. Flipping about the y axis with OS_SpriteOp 47 or *SFlipY will make it look back to front.

Remove wastage

If a sprite is not a whole number of words wide, it is possible that part of each row on the left and right is 'wasted'; that is, it does not form part of the sprite image. To remove this wastage, OS_SpriteOp 54 will align the sprite with the left hand side. If more than 32 free bits are on the right of the sprite, then these words will be removed.

Appending

Sprites can be tacked together, either horizontally or vertically, using OS_SpriteOp 35. No extra memory is used to do this.

Reading and altering pixels

To check the size of a sprite, OS_SpriteOp 40 will return its width, height, screen mode and whether it has a mask or not.

If you wish to read a pixel in a sprite, then OS_SpriteOp 41 will return colour and tint for a given x and y coordinate in the sprite. To write a pixel colour, OS_SpriteOp 42 must be used. It is given the coordinates, colour and tint to use.

Reading and altering the mask

Similar to these last two SWIs, OS_SpriteOp 43 will read a mask bit and OS_SpriteOp 44 will write it. Remember that a mask has the same number of bits per pixel as the image, but that the bits for each pixel must either be all set, or all clear.

VDU output to sprites

The VDU drivers can be directed to put their output into a sprite instead of the screen. OS_SpriteOp 60 will switch output to a sprite or to the screen. OS_SpriteOp 61 will switch output to a mask or the screen.

The save area described earlier is used by these calls. The space required for a save area can be determined by calling OS_SpriteOp 62.

Plotting sprites

To plot a sprite on the screen, OS_SpriteOp 28 and 34 are the simplest to use. They plot the sprite at the current graphics cursor position, using the current GCOL action. OS_SpriteOp 48 and 49 are similar, but the coordinates and GCOL action are instead passed explicitly.

Scaled and transformed plotting

A sprite can be plotted at any magnification using OS_SpriteOp 50 and 52.

Like these SWIs, OS_SpriteOp 53 will plot a sprite using scale factors and a translation table, but it uses the anti-aliased colour technique that the font manager uses for characters.

OS_SpriteOp 51 will paint a character onto the screen using scale factors.

OS_SpriteOp 55 and 56 will plot a sprite or mask with a linear transformation, such as a shear, stretch or reflection.

Format of a sprite area

The format of a sprite area is as follows:

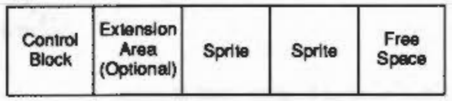


Figure 22.1 Format of a sprite area

The sprite area control block contains the following:

Word	**Contents**
1	Byte offset to last byte+1 (ie total size of sprite area)
2	Number of sprites in area
3	Byte offset to first sprite
4	Byte offset to first free word (ie byte after last sprite)
5...	Extension words (usually null)

The above offsets are relative to the start of the sprite area control block.

The format of the file created by a *ScreenSave or *SSave command is the same as a sprite area but without word 1 of the control block. This is because it is only valid in memory.

Format of a sprite

The format of a sprite is as follows:

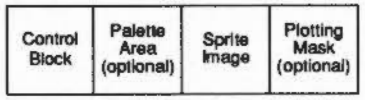


Figure 22.2 Format of a sprite

The Sprite Control Block contains the following:

Word	**Content**
1	Offset to next sprite
2 - 4	Sprite name, up to 12 characters with trailing zeroes
5	Width in words -1
6	Height in scan lines -1
7	First bit used (left end of row)
8	Last bit used (right end of row)
9	Offset to sprite image
10	Offset to transparency mask or offset to sprite image if no mask
11	Mode sprite was defined in
12...	Palette data (optional)

The size of the palette data block depends on the number of bits per pixel in the sprite's mode, since there will be one entry for each potential logical colour. 256 colour modes are the exception to this rule, because there are only 16 palette registers.

Note that 256 colour sprites created by *ScreenSave actually have 64 palette entries; the last 16 are the ones that are actually enforced.

Each entry is two words long. These are the words returned from OS_ReadPalette (SWI &2F). The format of these words is described with this SWI on page 2-209.

Format of a sprite image

The format of a sprite image is as follows:

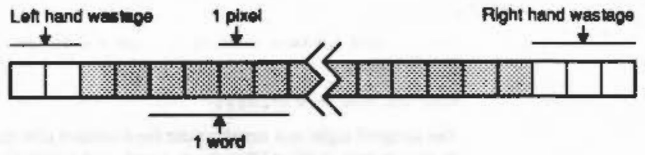


Figure 22.3 Format of a sprite image

The image contains the rows of the sprite from top to bottom, all word-aligned. Each pixel is a group of *bytes per character* bits (see OS_ReadVduVariables (SWI &31) on page 2-211). The least significant pixel in a word is the left-most one on the screen.

Note that in the diagram above, bit 0 of each word has been shown on the **left**, and bit 31 has been shown on the **right**; this is to clarify how wastage occurs. Note also that there will not necessarily be 4 pixels per word.

Format of a sprite mask

A sprite mask is the same size as the corresponding sprite image, and the same bits refer to each pixel. In the mask, the bits of each pixel must either all be set (the sprite's pixel is solid) or all be cleared (the pixel is transparent).

VDU commands

There are ways of selecting a sprite so that it can subsequently be used by the VDU commands described below to plot sprites.

The VDU commands are included for compatibility only and in RISC OS are of very little use since they only allow access to the system sprite area, whereas you will more likely be using user sprite areas.

Any programs being written for the Wimp must not use these VDU commands because there is only one location storing the setting for the selected sprite, not one per process.

As well as *SChoose and OS_SpriteOp 24, a sprite can be selected for VDU use by:

```
VDU 23,27,m,n|
```

where:	*m* = 0	is equivalent to *SChoose *n*
	m = 1	is equivalent to *SGet *n*

Plotting a sprite

Once a sprite has been selected by either of the three techniques above, it can be plotted using:

```
VDU 25,232 - 239,x;y;
```

The range of eight plot numbers are the standard plot options as defined in VDU 25 in the chapter entitled *VDU drivers*. *x* and *y* are in OS coordinates.

Service Calls

Service_SwitchingOutputToSprite (Service Call &72)

Output switched to sprite, mask or screen

On entry

R0 = &72 (reason code)

On exit

All registers preserved

Use

Issued when output is switched from and to a sprite immediately after the output is switched.

This service call should not be claimed.

SWI Calls

OS_SpriteOp (SWI &2E)

Controls the sprite system

On entry

R0 = reason code
Other registers depend on reason code

On exit

R0 preserved
Other registers depend on reason code

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant (RISC OS 2.0); SWI is re-entrant (RISC OS 2.5)

Use

This call controls the sprite system. It is indirected through SpriteV.

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

R0	Action
2	Screen save
3	Screen load
8	Read area control block
9	Initialise sprite area
10	Load sprite file
11	Merge sprite file
12	Save sprite file
13	Return name
14	Get sprite
15	Create sprite
16	Get sprite from user co-ordinates
24	Select sprite
25	Delete sprite
26	Rename sprite
27	Copy sprite
28	Put sprite
29	Create mask
30	Remove mask
31	Insert row
32	Delete row
33	Flip about x axis
34	Put sprite at user coordinates
35 *	Append sprite
36 *	Set pointer shape
37	Create/remove palette
40	Read sprite information
41	Read pixel colour
42	Write pixel colour
43	Read pixel mask
44	Write pixel mask
45	Insert column
46	Delete column
47	Flip about y axis
48	Plot sprite mask
49	Plot mask at user coordinates
50	Plot mask scaled
51 *	Paint character scaled
52 *	Put sprite scaled
53 *	Put sprite grey scaled
54	Remove lefthand wastage
55 *	Plot mask transformed
56	Put sprite transformed
57	Insert/delete rows
58	Insert/delete columns
60	Switch output to sprite
61	Switch output to mask
62	Read save area size

For details of each of these reason codes, see below.

Note that the reason codes marked with an asterisk are provided by the SpriteExtend module, which must be loaded for them to work.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 2 (SWI &2E)

Screen save

On entry

R0 = 2
R2 = pointer to pathname
R3 = palette flag (0 not to save, 1 to save)

On exit

R0, R2, R3 preserved

Use

This saves the current graphics window as a sprite file. The file contains a single sprite called 'screendump'. If R3 is 0, no palette information is saved with the file; if it is 1, the current palette is saved. It is equivalent to *ScreenSave.

See reason code 3 to reverse the operation and load a screen.

Related SWIs

OS_SpriteOp 3 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 3 (SWI &2E)

Screen load

On entry

R0 = 3
R2 = pointer to pathname

On exit

R0, R2 preserved

Use

This plots a sprite directly from a file to the screen. It changes mode if necessary and sets the palette to the setting held in the file. The sprite is plotted at the bottom left of the graphics window. After a mode change, this is the bottom left-hand corner of the screen. It is equivalent to *ScreenLoad.

See reason code 2 to reverse the operation and save a screen.

Related SWIs

OS_SpriteOp 2 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 8 (SWI &2E)

Read area control block

On entry

R0 = 8
R1 = pointer to control block of sprite area

On exit

R0, R1 preserved
R2 = total size of sprite area in bytes
R3 = number of sprites in area
R4 = byte offset to the first sprite
R5 = byte offset to the first free word

Use

This returns all the information contained in the control block of a sprite area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 9 (SWI &2E)

Initialise sprite area

On entry

R0 = 9
R1 = pointer to control block of sprite area

On exit

R0, R1 preserved

Use

This initialises a sprite area. It is equivalent to *SNew when used with the system area.

If you are initialising a user sprite area, then you must first initialise two words in the area header:

Address	Contents of word
area + 0	total size of area
area + 8	offset to first sprite (= 16, if the extension area is null)

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 10 (SWI &2E)

Load sprite file

On entry

R0 = 10
R1 = pointer to control block of sprite area
R2 = pointer to pathname

On exit

R0 - R2 preserved

Use

This loads the sprite definitions contained in the file into the sprite area, overwriting any definitions stored there already. It is equivalent to *SLoad when used with the system area.

The first word of the sprite area must be initialised to its size before you call this SWI.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 11 (SWI &2E)

Merge sprite file

On entry

R0 = 11
R1 = pointer to control block of sprite area
R2 = pointer to pathname

On exit

R0 - R2 preserved

Use

This merges the sprite definitions contained in the file with those in the sprite area. It is equivalent to *SMerge when used with the system area.

Note that there must be enough free space in the sprite area to hold both the new file and the original sprites, since it is only after the new file has been loaded that any of the original sprites are replaced by new ones that have the same name.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 12 (SWI &2E)

Save sprite file

On entry

R0 = 12
R1 = pointer to control block of sprite area
R2 = pointer to pathname

On exit

R0 - R2 preserved

Use

This saves the contents of a sprite area to a file. It is equivalent to *SSave when used with the system area.

The first word of the sprite area (its size) is not saved.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 13 (SWI &2E)

Return name

On entry

R0 = 13
R1 = pointer to control block of sprite area
R2 = pointer to buffer
R3 = maximum name length (ie buffer size)
R4 = sprite number (position in workspace – the first one is numbered 1)

On exit

R0 - R2 preserved
R3 = name length
R4 preserved

Use

This returns the name of the sprite whose position in the workspace (eg 3 for the third sprite) is given in R4. The name is placed in the buffer pointed to by R2 as a null-terminated string, the length of which is returned in R3.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 14 (SWI &2E)

Get sprite

On entry

R0 = 14 (&0E)
R1 = pointer to control block of sprite area
R2 = pointer to sprite name
R3 = palette flag (0 to exclude palette data, 1 to include it)

On exit

R0, R1 preserved
R2 = address of sprite (if in user sprite area)
R3 preserved

Use

This defines the sprite identified to be the current contents of an area of the screen. It is delimited by the current and old cursor positions (inclusive). If the sprite already exists, it is overwritten. It is equivalent to *SGet when used with the system area.

Any part of the designated area which lies outside the current graphics window is filled with the current background colour in the sprite.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 2-251. You must not call this SWI with bit 9 of R0 set; that is, R2 must always point to a sprite name.

Related SWIs

OS_SpriteOp 16 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 15 (SWI &2E)

Create sprite

On entry

R0 = 15 (&0F)
R1 = pointer to control block of sprite area
R2 = pointer to sprite name
R3 = palette flag (0 to exclude palette data, 1 to include it)
R4 = width in pixels
R5 = height in pixels
R6 = mode number

On exit

R0 - R6 preserved

Use

This creates a blank sprite of a given size.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 2-251. You must not call this SWI with bit 9 of R0 set; that is, R2 must always point to a sprite name.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 16 (SWI &2E)

Get sprite from user coordinates

On entry

R0 = 16 (&10)
R1 = pointer to control block of sprite area
R2 = pointer to sprite name
R3 = palette flag (0 to exclude palette data, 1 to include it)
R4 = left hand edge OS screen coordinate (inclusive)
R5 = bottom edge OS screen coordinate (inclusive)
R6 = right hand edge OS screen coordinate (inclusive)
R7 = top edge OS screen coordinate (inclusive)

On exit

R0, R1 preserved
R2 = address of sprite (if in user sprite area)
R3 - R7 preserved

Use

This picks up an area of the screen, which is delimited by the coordinates supplied (inclusive), as a sprite. If the sprite already exists, it is overwritten.

Any part of the designated area which lies outside the current graphics window is filled with the current background colour in the sprite.

Setting bit 8 or 9 of R0 alters the interpretation of R1 – for a description see the section entitled *Common parameters* on page 2-251. You must not call this SWI with bit 9 of R0 set; that is, R2 must always point to a sprite name.

Related SWIs

OS_SpriteOp 14 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 24 (SWI &2E)

Select sprite

On entry

R0 = 24 (&18)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0, R1 preserved
R2 = address of sprite (if in user sprite area), otherwise preserved

Use

Select a particular sprite for subsequent plotting. That is, the VDU 25,232-239 commands will use the selected sprite. It is equivalent to *SChoose when used with the system area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 25 (SWI &2E)

Delete sprite

On entry

R0 = 25 (&19)
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - R2 preserved

Use

This deletes the definition of a particular sprite. It is equivalent to *SDelete when used with the system area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 26 (SWI &2E)

Rename sprite

On entry

R0 = 26 (&1A)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = pointer to new name

On exit

R0 - R3 preserved

Use

This changes the name of a sprite. An error is produced if a sprite of the new name already exists in the same sprite area. It is equivalent to *SRename when used with the system area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 27 (SWI &2E)

Copy sprite

On entry

R0 = 27 (&1B)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = pointer to new name

On exit

R0 - R3 preserved

Use

This copies a sprite within a sprite area. An error is produced if a sprite of the new name already exists in the same sprite area. It is equivalent to *SCopy when used with the system area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 28 (SWI &2E)

Put sprite

On entry

R0 = 28
R1 = pointer to control block of sprite area
R2 = sprite pointer
R5 = plot action

On exit

R0 - R2, R5 preserved

Use

This plots the sprite identified with its bottom left corner at the current graphics cursor position using the plot action specified in R5:

Value	Action
0	Overwrite colour on screen
1	OR with colour on screen
2	AND with colour on screen
3	exclusive OR with colour on screen
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND with colour on screen with NOT of sprite pixel colour
7	OR with colour on screen with NOT of sprite pixel colour
&08	If set, then use the mask, otherwise don't
&10	ECF pattern 1
&20	ECF pattern 2
&30	ECF pattern 3
&40	ECF pattern 4
&50	Giant ECF pattern (patterns 1 - 4 placed side by side)

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 48 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 29 (SWI &2E)

Create mask

On entry

R0 = 29
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - R2 preserved

Use

This creates a mask for the specified sprite with all pixels set to be solid.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 30 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 30 (SWI &2E)

Remove mask

On entry

R0 = 30
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - R2 preserved

Use

This removes the mask definition for a given sprite.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 29 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 31 (SWI &2E)

Insert row

On entry

R0 = 31
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = row number

On exit

R0 - R3 preserved

Use

This inserts a row in the sprite at the position identified, shifting all rows above it up one. All pixels in the new row are set to colour zero, or to transparent if the sprite has a mask. Rows are numbered from the bottom upwards with the bottom row being number zero. If the row number is equal to the height of the sprite it will go on top. Any value above this will generate an error.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 32, 45 and 46 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 32 (SWI &2E)

Delete row

On entry

R0 = 32
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = row number

On exit

R0 - R3 preserved

Use

This deletes a row in the sprite at the position identified, shifting all rows above it down one. Rows are numbered from the bottom upwards with the bottom row being number zero. If the row number is greater than or equal to the height of the sprite it will generate an error.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 31, 45 and 46

Related vectors

SpriteV

OS_SpriteOp 33 (SWI &2E)

Flip about x axis

On entry

R0 = 33
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - R2 preserved

Use

This takes the sprite identified and reflects it about the x axis so that it is upside down. Thus, its top row on entry becomes the bottom row on exit, and so on.

It is equivalent to *SFlipX when used on the system area sprites.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 47 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 34 (SWI &2E)

Put sprite at user coordinates

On entry

R0 = 34
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate
R4 = y coordinate
R5 = plot action

On exit

R0 - R5 preserved

Use

This plots a sprite at the external coordinates supplied, using the plot action supplied in R5:

Value	Action
0	Overwrite colour on screen
1	OR with colour on screen
2	AND with colour on screen
3	exclusive OR with colour on screen
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND with colour on screen with NOT of sprite pixel colour
7	OR with colour on screen with NOT of sprite pixel colour
&08	If set, then use the mask, otherwise don't
&10	ECF pattern 1
&20	ECF pattern 2
&30	ECF pattern 3
&40	ECF pattern 4
&50	Giant ECF pattern (patterns 1 - 4 placed side by side)

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 35 (SWI &2E)

Append sprite

On entry

R0 = 35
R1 = pointer to control block of sprite area
R2 = sprite pointer 1
R3 = sprite pointer 2
R4 = 0 to merge horizontally, or 1 to merge vertically

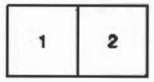
On exit

R0 - R4 preserved

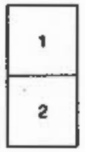
Use

This call can be used to merge two sprites of the same height or width into one sprite, tacking them together vertically or horizontally.

The sprites are appended horizontally in the following order:



The sprites are appended vertically in the following order:



The result of the merge is stored in sprite 1 and sprite 2 is deleted. Thus the merge does not consume any extra memory.

Attempting to merge two sprites with different vertical or horizontal sizes will result in an error.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 36 (SWI &2E)

Set pointer shape

On entry

R0 = 36
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = bitfield (see below)
R4 = x offset of active point
R5 = y offset of active point
R6 = scale factors (0 to scale for the mode)
R7 = pixel translation table

On exit

R0 - R3 preserved

Use

This call sets any of the hardware pointer shapes to be programmed from a sprite, with some degree of mode independence – ie the aspect ratio is catered for.

Note that in high resolution monochrome modes (eg mode 23), the pointer shape resolution is four times worse horizontally than the pixel resolution, and only colours 0, 1 and 3 can be used in the pointer shape definition. This call will cater for this problem by halving the width of the pointer, so that it is still possible to see what it is, although the pointer will be twice as wide as usual.

R3 on entry is a bitfield composed of the following fields:

Bit	Meaning
0-3	pointer shape number, currently in the range 1 - 4
4	if clear, then set the pointer shape data
5	if clear, then set the palette from the sprite
6	if clear, then program the pointer shape number

Bits 4, 5, and 6 of this bitfield can be used to defer certain aspects of this call until later. For example, if you wanted to set up the pointer shape without displaying the pointer, bits 5 and 6 would be set.

The coordinates in R4 and R5 are relative pixels from the top left corner of the sprite.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_Word 8 (SWI &07), Wimp_SetPointerShape (SWI &400D8)

Related vectors

SpriteV

OS_SpriteOp 37 (SWI &2E)

Create/remove palette

On entry

R0 = 37
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = sub-reason code:
- −1 ⇒ read current palette size
- 0 ⇒ remove palette from sprite
- otherwise ⇒ create palette in sprite

On exit

R0 - R2 preserved
R3 = size of palette or 0 if none (if R3 = −1 on entry); else preserved
R4 = poinyter to palette or 0 if none (if R3 = −1 on entry)
R5 = mode (if R3 = −1 on entry)

Use

This call creates a palette, removes a palette, or finds the size of the palette associated with a given sprite.

If you add or remove a sprite's palette when output is switched to the sprite you will invalidate the current display pointers In such cases you should switch output away from the sprite, modify the palette, and then switch output back to the sprite.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 40 (SWI &2E)

Read sprite information

On entry

R0 = 40
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - R2 preserved
R3 = width in pixels
R4 = height in pixels
R5 = mask status (0 for no mask, 1 for mask)
R6 = screen mode in which the sprite was defined

Use

This returns information about the sprite, giving its width and height in pixels, whether the sprite has a mask and the screen mode in which the sprite was defined.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 41 (SWI &2E)

Read pixel colour

On entry

R0 = 41
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate (in pixels)
R4 = y coordinate (in pixels)

On exit

R0 - R4 preserved
R5 = colour
R6 = tint

Use

Given x and y coordinates in R3 and R4 (in pixels relative to the bottom left of the sprite definition), this call returns the current colour of the pixel at that position.

The colour and tint returned depends on the mode. If it is not a 256 colour mode, then colour is from zero to the number of colours–1 and tint is zero. In 256 colour modes, the colour is from 0 to 63 and tint is either 0, 64, 128 or 192.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 42 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 42 (SWI &2E)

Write pixel colour

On entry

R0 = 42
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate
R4 = y coordinate
R5 = colour
R6 = tint

On exit

R0 - R6 preserved

Use

Given x and y coordinates (in pixels from the bottom left of the sprite definition), and colour and tint in R5 and R6, this call sets the pixel at the position given to that colour.

The colour and tint values used depend on the mode. If it is not a 256 colour mode, then colour is from zero to the number of colours–1 and tint is ignored. In 256 colour modes, the colour is from 0 to 63 and tint is either 0, 64, 128 or 192: ie only bits 6 and 7 are used.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 41 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 43 (SWI &2E)

Read pixel mask

On entry

R0 = 43
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate
R4 = y coordinate

On exit

R0 - R4 preserved
R5 = mask status (0 = transparent, 1 = solid)

Use

Given x and y coordinates in R3 and R4 (in pixels relative to the bottom left of the sprite definition), this call returns the current state of the mask at that position.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 44 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 44 (SWI &2E)

Write pixel mask

On entry

R0 = 44
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate
R4 = y coordinate
R5 = mask status (0 = transparent, 1 = solid)

On exit

R0 - R5 preserved

Use

Given x and y coordinates (in pixels from the bottom left of the sprite definition), and mask state in R5, this call sets the pixel at the position given to that mask.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 43 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 45 (SWI &2E)

Insert column

On entry

R0 = 45
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = column number

On exit

R0 - R3 preserved

Use

This inserts a column at the position identified, shifting all columns after it one place to the right. The new column is set to have either transparent or colour zero pixels, depending on whether the sprite has a mask or not. Columns are numbered from the left with the left-hand one being number zero.

If the column number is equal to the width of the sprite it will go after the right hand side. Any value above this will generate an error.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 31, 32 and 46 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 46 (SWI &2E)

Delete column

On entry

R0 = 46
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = column number

On exit

R0 - R3 preserved

Use

This deletes a column from the position identified, shifting all columns after it one place to the left. Columns are numbered from the left with the left-hand one being number zero.

If the column number is greater than or equal to the width of the sprite it will generate an error.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 31, 32 and 45 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 47 (SWI &2E)

Flip about y axis

On entry

R0 = 47
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - R2 preserved

Use

This takes the sprite identified and reflects it about the y axis so that it is facing in the opposite direction. Thus, its leftmost column on entry becomes the rightmost column on exit, and so on.

It is equivalent to *SFlipY when used with the system sprite area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 33 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 48 (SWI &2E)

Plot sprite mask

On entry

R0 = 48
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - R2 preserved

Use

This plots a sprite mask in the background colour and action with its bottom left corner at the graphics cursor position. That is, all 1 bits in the mask are plotted in the background colour and action, and all 0 bits are ignored. If the sprite has no mask, a solid rectangle the same size as the sprite is drawn in the current background colour and action (as if there was a mask which was completely solid).

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 28 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 49 (SWI &2E)

Plot mask at user coordinates

On entry

R0 = 49
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate
R4 = y coordinate

On exit

R0 - R4 preserved

Use

This plots in the background colour and action through a sprite mask at the external coordinates supplied.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 48 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 50 (SWI &2E)

Plot mask scaled

On entry

R0 = 50
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = *x* coordinate to plot at
R4 = y coordinate to plot at
R6 = scale factors

On exit

R0 - R6 preserved

Use

A sprite mask is plotted on the screen, using the current background colour and action and the scaling factors provided.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 51 (SWI &2E)

Paint character scaled

On entry

R0 = 51
R1 = character code
R3 = *x* coordinate to plot
R4 = y coordinate to plot
R6 = scale factors

On exit

R0, R1, R3, R4, R6 preserved

Use

The specified character is plotted on the screen with its lower left hand corner at the specified coordinate, using the current graphics foreground colour and action.

See the technical description in this chapter for a description of plot actions.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 52 (SWI &2E)

Put sprite scaled

On entry

R0 = 52
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate to plot
R4 = y coordinate to plot
R5 = plot action
R6 = scale factors
R7 = pixel translation table

On exit

R0 - R7 preserved

Use

This will plot a sprite on the screen using:

- the coordinate specified by R3 and R4
- the plot action specified by R5.
- the scale factor specified by R6
- the pixel translation table pointed to by R7

The plot actions specified in R5 are:

Value	Action
0	Overwrite colour on screen
1	OR with colour on screen
2	AND with colour on screen
3	exclusive OR with colour on screen
4	Invert colour on screen
5	Leave colour on screen unchanged
6	AND with colour on screen with NOT of sprite pixel colour
7	OR with colour on screen with NOT of sprite pixel colour
&08	If set, then use the mask, otherwise don't
&10	ECF pattern 1
&20	ECF pattern 2
&30	ECF pattern 3
&40	ECF pattern 4
&50	Giant ECF pattern (patterns 1 - 4 placed side by side)

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 53 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 53 (SWI &2E)

Put sprite grey scaled

On entry

R0 = 53
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x coordinate to plot at
R4 = y coordinate to plot at
R5 = 0
R6 = scale factors
R7 = pixel translation table

On exit

R0 - R7 preserved

Use

This call is similar to OS_SpriteOp 52, except that it performs anti-aliasing on the sprite as it scales it. This is the same technique that the Font Manager uses on characters. This means that the sprite must have been defined in a 4 bits per pixel mode (16 colours), and the pixels must reflect a linear grey scale, as with anti-aliased font definitions.

This call is considerably slower than OS_SpriteOp 52 (Put sprite scaled) and should only be used when the quality of the image is of the utmost importance. To speed up redrawing of an anti-aliased sprite, it is possible to draw the image into another sprite (using OS_SpriteOp 60 – switch output to sprite), which can then be redrawn more quickly.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 52 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 54 (SWI &2E)

Remove left hand wastage

On entry

R0 = 54
R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit

R0 - R2 preserved

Use

In general, sprites have a number of unused bits in the words corresponding to the left and right hand edges of each pixel row. This call removes the left hand wastage, so that the left hand side of the sprite is word aligned.

The right hand wastage is increased by the number of bits that were removed. If this is now more than 32 bits then a whole word is removed from each row of the sprite, and the rest of the sprite area moved down to fill the gap.

Note that when you switch output to a sprite using OS_SpriteOp 60 or 61, the left-hand wastage is also removed.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 60 and 61 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 55 and 56 (SWI &2E)

Transformed sprite handling

On entry

R0 = 55 (PlotMaskTransformed) or 56 (PutSpriteTransformed)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = flag word:
    bit 0 set ⇒ R6 = pointer to destination coordinates, else matrix
    bit 1 set ⇒ R4 = pointer to source rectangle inside sprite
    bits 2-31 reserved (must be 0)
R4 = pointer to source rectangle coordinate block (if R3 bit 1 set):
    R4+0,4 = x0, y0 one corner in sprite (in pixels)
    R4+8,12 = x1,y1 second corner in sprite (in pixels)
R5 = GCOL action (for PutSpriteTransformed)
    +8 if mask is to be used
R6 = pointer to matrix (if R3 bit 0 clear):
    R6+0,4,8,12,16,20 = matrix (as for Draw module)
R6 = pointer to destination coordinate block (if R3 bit 0 set):
    R6+0,4 = X0,Y0 on screen (1/256th OS unit)
    R6+8,12 = X1,Y1 on screen (1/256th OS unit)
    R6+16,20 = X2,Y2 on screen (1/256th OS unit)
    R6+24,28 = X3,Y3 on screen (1/256th OS unit)
R7 = pointer to translation table (≤ 0 ⇒ none)

On exit

—

Use

This call is not available in RISC OS 2.0.

The source coordinates are inclusive at the bottom-left, and exclusive at the top-right. If no source rectangle is given, the default is x0 = 0, x1 = width of sprite (in pixels), y0 = height of sprite (in pixels), and y1 = 0. Note that the y coordinates are the reverse of what you might expect.

When specifying a destination parallelogram, the source rectangle is mapped onto the destination as follows:

x0,y0	X0,Y0
x1,y0	X1,Y1
x1,y1	X2,Y2
x0,y1	X3,Y3

In future it may be possible to set the destination to an arbitrary quadrilateral, rather than a parallelogram. In order to reserve this possibility, the current version returns an error if the destination is not a parallelogram.

For PutSpriteTransformed, the sprite is plotted through its mask only if it both has one, and bit 3 of R5 is set. R5 is ignored for PlotMaskTransformed.

The SWI returns an error if any of R3 bits 2 - 31 are set, to ensure that these are left clear by software developers.

The SWI covers exactly those pixels on the screen that a call to Draw_Fill would produce for a rectangle of the same size with the same transformation matrix, where it is filling to half-way through the boundary.

When plotting using a destination parallelogram, the source rectangle must be entirely within the sprite. For plotting with a matrix, the source rectangle will be clipped to the sprite boundaries prior to transformation.

If the source rectangle (after clipping, if using a matrix) has no area, i.e. x0 = x1 OR y0 = y1 then an error will be generated, as it is not possible to choose a colour in which to fill the destination.

Note that the SWI does allow x0>x1 or y0>y1 or both. When plotting with a matrix there is no difference between x0 and x1 swapped, or y0 and y1 swapped, but when specifying a destination parallelogram the image will be reflected.

Due to the mechanism of the routine the accuracy is not absolute. The SWI will always cover the same area as a Draw filled path, but not necessarily with the right source pixel data from the sprite. The worst possible error (in a fraction of a source pixel) at one end of the plotted area is given by *destination width or height*/65536.

The table below gives more information on the maximum errors attainable:

Destination size	Worst possible error in source pixels
5	0.0000763
10	0.0001526
50	0.0007629
100	0.0015259
500	0.0076294
1000	0.0152588
5000	0.0762939
10000	0.1525879

(The largest output possible is 32767 pixels)

For example, when plotting a sprite to a destination width of 5000 pixels, the worst error possible in the position in the source rectangle of the final pixel plotted is about 1/13 of a source pixel.

Note that if these errors (usually too small to notice) must be avoided then the sprite should be plotted in parts – perhaps by dividing the plotting into four areas.

Errors:

Attempt to set reserved flags

RO bits 2 - 31 must be zero.

Source rectangle area zero

The area of the source rectangle must be non-zero, so the sprite routine(s) will have some valid colour with which to plot the output.

Source rectangle not inside sprite

The source rectangle must be totally inside the sprite.

SpriteExtend can only do linear transformations.

The current version of the transformation routines can only perform linear transformations, and not any arbitrary rotation.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 57 and 58 (SWI &2E)

Insert/delete rows/columns from a sprite

On entry

R0 = 57 (InsertDeleteRows) or 58 (InsertDeleteColumns)
R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = row/column to start deletion at or to insert before
R4 = number of rows/columns to insert (if +ve) or delete (if -ve)

On exit

R0 - R4 preserved

Use

This call is not available in RISC OS 2.0.

For insertion R4 > 0, and R3 specifies the row or column to insert before. For a sprite of *n* rows × *m* columns the rows are numbered from 0 at the bottom to *n*–1 at the top, and columns from 0 at the left to *m*–1 at the top. Thus to insert rows/columns on the edges of the sprite:

R0	R3	Insertion point
57	0	bottom edge (ie before the first row)
	n	top edge (ie before the row beyond the last row)
58	0	left edge (ie before the first column)
	m	right edge (ie before the column beyond the last column)

The inserted rows/columns are set to colour 0. If the sprite has a mask then rows/columns are inserted into that as well, and the inserted area is transparent.

For deletion R4 < 0, and R3 specifies the first row or column to be deleted. The rows/columns from R3 to (R3–R4–1) will be deleted,. An error will be given if R3 or R4 are out of range for the sprite.

Related SWIs

None

Related vectors

SpriteV

OS_SpriteOp 60 (SWI &2E)

Switch output to sprite

On entry

R0 = 60
R1 = pointer to control block of sprite area
R2 = sprite pointer to switch to sprite or 0 to switch to screen
R3 = save area:
  0 = no save area
  1 = system save area
  any other value = pointer to save area

On exit

R0 preserved
R1 = previous value
R2 = previous value
R3 = previous value

Use

This call can cause VDU calls to be sent to the screen memory, or to a sprite's image.

R2 has its usual function as a sprite pointer or it can be zero. If it is a sprite pointer, then this call will switch VDU output to a sprite. If it is a zero, then this call will switch output to the screen.

The save area can have a number of values. If it is zero, then no save area will be used. If it is one, then the system save area is used, which is the save area used by RISC OS when output is directed to the screen. You should not use the system save area yourself if you wish to preserve the VDU output state for the screen. Any other value of R3 is considered to be a pointer to the save area.

If the first word of the save area is zero, then the VDU state will be initialised to suitable defaults for the given sprite's mode. When output is switched away from the sprite, the current VDU state is copied into the save area, and the first word is overwritten with a non-zero value. If output is subsequently switched back to the sprite, the VDU state will be restored from the save area.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 61 and 62 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 61 (SWI &2E)

Switch output to mask

On entry

R0 = 61
R1 = pointer to control block of sprite area
R2 = sprite pointer to switch to mask or 0 to switch to screen
R3 = save area:
  0 = no save area
  1 = system save area
  any other value = pointer to save area

On exit

R0 preserved
R1 = previous value
R2 = previous value
R3 = previous value

Use

This call can cause VDU calls to be sent to the screen memory, or to a sprite's mask.

A sprite's mask has the same number of bits per pixel as its image, where a value of 0 is a transparent pixel and a value of all 1's represents a solid pixel. For example, &0F for 4 bits per pixel. Other values are not permitted.

See OS_SpriteOp 60 for a general description of how this call works.

Note that when plotting into a sprite's mask, the only colours that should be used are 0 and (number of colours – 1), that is:

- in 2 colour modes use colours 0 and 1
- in 4 colour modes use colours 0 and 3
- in 16 colour modes use colours 0 and 15
- in 256 colour modes use colour 0 tint 0, and colour 63 tint 192 (&C0)

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 60 and 62 (SWI &2E)

Related vectors

SpriteV

OS_SpriteOp 62 (SWI &2E)

Read save area size

On entry

R0 = 62
R1 = pointer to control block of sprite area
R2 = sprite pointer, or 0 for the screen

On exit

R0 - R2 preserved
R3 = size of required save area in bytes

Use

This calls calculates how large a save area must be for a given sprite. Remember that a save area must be word aligned.

Setting bit 8 or 9 of R0 alters the interpretation of R1 and R2 – for a description see the section entitled *Common parameters* on page 2-251.

Related SWIs

OS_SpriteOp 60 and 61 (SWI &2E)

Related vectors

SpriteV

*Commands

*Configure SpriteSize

Sets the configured amount of memory reserved for the system sprite area

Syntax

```
*Configure SpriteSize mK|n
```

Parameters

mK	number of kilobytes of memory reserved
n	number of pages of memory reserved; n ≤ 127

Use

*Configure SpriteSize sets the configured amount of memory reserved for the system sprite area. If you pass a parameter of 0, then no space is reserved for system sprites. The default value is one page of memory.

You can also use OS_ChangeDynamicArea (SWI &2A) to alter dynamically the system sprite size. For more information, refer to the chapter entitled *Memory Management*.

The change takes effect on the next hard reset.

Example

```
*Configure SpriteSize 20K
```

Related commands

None

Related SWIs

None

Related vectors

None

*SChoose

Selects a sprite for use in subsequent sprite plotting operations

Syntax

```
*SChoose sprite_name
```

Parameters

sprite_name	name of a sprite in the system sprite area

Use

*SChoose selects a sprite from the system sprite area for use in subsequent sprite plotting operations. It is used in conjunction with VDU 25,232-239 operations. You should see the warning in the *Technical Details* about using these obsolescent VDU calls.

The sprite name is not case-sensitive.

Example

```
*SChoose fish
```

Related commands

None

Related SWIs

OS_SpriteOp 24 (SWI &2E)

Related vectors

SpriteV

*SCopy

Makes a copy of a sprite within the system sprite area

Syntax

```
*SCopy source_sprite_name dest_sprite_name
```

Parameters

source_sprite_name	name of source sprite in the system sprite area
dest_sprite_name	name of destination sprite (to be placed in the system sprite area)

Use

*SCopy makes a copy of the source sprite within the system sprite area, and renames it as the destination sprite. An error is generated if the destination sprite already exists.

Example

```
*SCopy acorn squirrel
```

Related commands

None

Related SWIs

OS_SpriteOp 27 (SWI &2E)

Related vectors

SpriteV

*ScreenLoad

Loads the contents of a sprite file into the graphics window

Syntax

```
*ScreenLoad filename
```

Parameters

filename	a valid pathname, specifying a sprite file

Use

*ScreenLoad loads the contents of a sprite file (saved, for example, with the *ScreenSave command) into the graphics window, which is typically the whole screen.

It changes mode if necessary and sets the palette to the setting in the file. The first sprite in the file is plotted at the bottom left hand corner of the graphics window. After a mode change, this is the bottom left hand corner of the screen.

Example

```
*ScreenLoad $.sprites.animals.koala
```

Related commands

*ScreenSave

Related SWIs

OS_SpriteOp 3 (SWI &2E)

Related vectors

None

*ScreenSave

Saves the contents of the graphics window and its palette to a file

Syntax

```
*ScreenSave filename
```

Parameters

filename	a valid pathname, specifying a file

Use

*ScreenSave saves the contents of the graphics window (typically the whole screen) and its palette to a file, which is saved as a sprite. The sprite file created will contain one sprite called 'screendump'.

You can then load this file into Paint or Draw.

Example

```
*ScreenSave My.Pic
```

Related commands

*ScreenLoad

Related SWIs

OS_SpriteOp 2 (SWI &2E)

Related vectors

None

*SDelete

Deletes one or more sprites from the system sprite area

Syntax

```
*SDelete sprite_name1 [sprite_name2…]
```

Parameters

sprite_name1	name of a sprite in the system sprite area
sprite_name2…	optional extra sprites to delete

Use

*SDelete deletes one or more sprites from the system sprite area.

If an error occurs (such as a sprite not existing) *SDelete will stop immediately, and no further sprites will be deleted.

Example

```
*SDelete fish cake elephant
```

Related commands

None

Related SWIs

OS_SpriteOp 25 (SWI &2E)

Related vectors

SpriteV

*SFlipX

Reflects a sprite in the system sprite area about its x axis

Syntax

```
*SFlipX sprite_name
```

Parameters

sprite_name	name of a sprite in the system sprite area

Use

*SFlipX reflects a sprite in the system sprite area about its x axis so it is upside down.

Example

```
*SFlipX sloth
```

Related commands

*SFlipY

Related SWIs

OS_SpriteOp 33 (SWI &2E)

Related vectors

SpriteV

*SFlipY

Reflects a sprite in the system area about its y axis

Syntax

```
*SFlipY sprite_name
```

Parameters

sprite_name	name of a sprite in the system sprite area

Use

*SFlipY reflects a sprite in the system sprite area about its y axis so it faces in the opposite direction.

Example

```
*SFlipY sloth
```

Related commands

*SFlipX

Related SWIs

OS_SpriteOp 47 (SWI &2E)

Related vectors

SpriteV

*SGet

Gets a sprite from the screen

Syntax

```
*SGet sprite_name
```

Parameters

sprite_name	name of new sprite in the system sprite area

Use

*SGet gets a sprite from a rectangular area of the screen, defined by the two most recent graphics positions (inclusive). It then saves this sprite in the system sprite area with the given name. If the sprite already exists, it is overwritten.

Any part of the designated area which lies outside the current graphics window is filled in the sprite with the current background colour.

Example

```
*SGet screenpart
```

Related commands

*ScreenSave

Related SWIs

OS_SpriteOp 14 (SWI &2E)

Related vectors

SpriteV

*SInfo

Displays information on the system sprite workspace

Syntax

```
*SInfo
```

Parameters

None

Use

*SInfo displays information on the system sprite workspace. It prints out the amount of system sprite workspace currently reserved, the amount of free space in that workspace and the number of sprites defined.

Example

```
*SInfo
Sprite status
  8 Kbytes sprite workspace
  7328 byte(s) free
  2 sprite(s) defined
```

Related commands

None

Related SWIs

OS_SpriteOp 8 (SWI &2E)

Related vectors

SpriteV

*SList

Lists the names of all the sprites in the system sprite area

Syntax

```
*SList
```

Parameters

None

Use

*SList lists the names of all the sprites in the system sprite area.

Example

```
*SList
!koala
!sloth
```

Related commands

None

Related SWIs

OS_SpriteOp 8 (SWI &2E)

Related vectors

SpriteV

*SLoad

Loads a sprite file into the system sprite area

Syntax

```
*SLoad filename
```

Parameters

filename	full name of file to load

Use

*SLoad loads a file containing sprite definitions into the system sprite area. If there is insufficient memory, then an error is given and nothing is loaded. Any sprites which are in memory when this command is given are lost.

Example

```
*SLoad $.sprites.animals.koala
```

Related commands

*ScreenLoad

Related SWIs

OS_SpriteOp 10 (SWI &2E)

Related vectors

SpriteV

*SMerge

Merges the sprites in a file with those in the system sprite area

Syntax

```
*SMerge filename
```

Parameters

filename	full name of file to load

Use

*SMerge merges the sprites in a file with those in the system sprite area. If there is insufficient memory, then an error is given and nothing is loaded. Any sprites in memory with the same name as any in the file are lost.

Note that there must be enough free space in the sprite area to hold both the new file and the original sprites, since it is only after the new file has been loaded that any of the original sprites are replaced by new ones that have the same name.

Example

```
*SMerge $.sprites.animals.koala
```

Related commands

None

Related SWIs

OS_SpriteOp 11 (SWI &2E)

Related vectors

SpriteV

*SNew

Deletes all the sprites in the system sprite area

Syntax

```
*SNew
```

Parameters

None

Use

*SNew deletes all the sprites in the system sprite area, and so frees all the sprite workspace.

Related commands

None

Related SWIs

OS_SpriteOp 9 (SWI &2E)

Related vectors

SpriteV

*SRename

Renames a sprite within the system sprite area

Syntax

```
*SRename old_sprite_name new_sprite_name
```

Parameters

old_sprite_name	name of a sprite in the system sprite area
new_sprite_name	new name of the sprite

Use

*SRename renames a sprite within the system sprite area. An error is generated if a sprite having the new name already exists.

A sprite name can contain any sequence of printable characters, other than a space; although upper-case letters will be changed to lower-case ones.

Example

```
*SRename thong flipflop
```

Related commands

None

Related SWIs

OS_SpriteOp 26 (SWI &2E)

Related vectors

SpriteV

*SSave

Saves the system sprite area as a sprite file

Syntax

```
*SSave filename
```

Parameters

filename	name of file to save

Use

*SSave saves all the sprites currently in the system sprite area as a sprite file. You can then load or merge the file later on.

Example

```
*SSave $.sprites.animals.koala
```

Related commands

*SLoad, *SMerge

Related SWIs

OS_SpriteOp 12 (SWI &2E)

Related vectors

SpriteV

23 Character Input

Introduction

The Character Input system can get characters from the computer's input devices. They can be any one of the following:

- the keyboard
- the serial port
- a file on any filing system

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

It provides a means of directing characters from the selected device to the program that requests them. It can also hold them, waiting until the program is ready to take them.

Overview

Before you read this chapter, you should have read the chapter entitled *Character Output* on page 2-1. In many ways, character input and output are one entity, which has been logically split in this manual. So there are some things which are mentioned there and not here that apply to both chapters.

Like character output, a stream system is used by character input. Here, you can select from one of three streams; keyboard, serial and file. Only one stream can be selected at once otherwise data coming from two places would get jumbled. Direct control of devices is available, especially in the case of the keyboard.

Streams

Any program taking input from the stream system doesn't have to know where characters are coming from. Most programs don't since it will not affect the way they run.

OS_ReadC

The core of the input stream is OS_ReadC which gets a single character from the currently selected input stream. It is in turn called by many other SWIs, OS_ReadLine (SWI &E) for example. This device independence makes programs much easier to write.

Buffers

Like character output, all input streams are buffered. Input devices are asynchronous to programs and must have their characters stored in a temporary place in memory until required. A good example of buffering in use is a terminal emulator program. It waits until something appears at the serial input buffer, then sends it to the VDU. At the same time, it waits until something appears in the keyboard buffer and sends it to the serial output buffer. Because of the buffering of inputs and outputs, the program can do all this at its own pace.

Keyboard

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

Keyboard handlers

The keyboard driver is actually two sections. One, which is fixed, handles the keyboard interrupt and low-level control. It feeds the raw code onto the second part, the keyboard handler.

The keyboard handler converts the keycode into an ASCII form, with extensions for special characters. This can be replaced by a custom version if required.

Basic operation

At a basic level, the keyboard works like this:

1 One or more keys are pressed, which cause an interrupt.
2 The keyboard driver gets a raw key number from the keyboard.
3 The raw key number is passed to the keyboard handler, where it is converted into a form more like the program expects. This can be:
   - an ASCII character.
   - a non-ASCII character, such as a function key or arrow.
   - a special key, such as Escape or Break that must be acted on immediately.
4 Apart from some special keys, this character is then stored in the keyboard buffer.

When a program wants a character from the input stream (in this example, the keyboard):

- When called by a program, the stream system gets the first character from the keyboard buffer (or waits if there is none there).
- Return the character to the program or perform the appropriate action if it is a function key, arrow, etc.

Advanced features

Also, there are a number of extra operations that the keyboard driver can perform:

- The interpretation of function keys, arrow keys and the numeric keypad can all be changed to various modes.
- The auto-repeat of keys can be adjusted, both the initial delay and the rate of repeat.
- The keyboard can be scanned directly, rather than going through any buffering.
- The keyboard handler can even be completely replaced with a custom handler.

About 30 SWIs and six * Commands exist purely for keyboard control. The section entitled *Technical Details* on page 2-342 covers how they work together.

Reset, Break and Escape

These three terms can become very confused, especially so when talking about the keyboard versus a program's view of the keyboard driver.

Reset

Reset is a unique key. Unlike all others it does not send a key code to the keyboard driver. It is connected to a separate line on the keyboard connector and physically resets the computer. This cannot be stopped by a program. When a reset occurs, some parts of the system are initialised.

There are three kinds of reset:

- A soft reset (with no other modifying keys) will initialise some parts of the system, but allow a lot to resume unaffected.
- If Shift is pressed at the same time, this is called a Shift-reset. This causes the machine to do a soft reset and then attempt an auto-boot from the default filing system (provided the computer and filing system have been configured for this using *Configure Boot).
- Ctrl-reset is a hard reset. This will initialise far more of the system and should only be necessary if something serious has occurred. It will put the computer into a 'just turned on' state in most cases.

BBC/Master users note that Reset is what used to be called Break on those machines.

Break

Break is a key. You can separately configure Break, Shift-Break, Ctrl-Break and Ctrl-Shift-Break to cause a reset, an escape condition or do nothing. By default:

- Break generates an escape condition
- Shift-Break causes a reset
- Ctrl-Break causes a reset
- Ctrl-Shift-Break causes a reset

Escape

Escape is a way of the user sending a signal to a program or its runtime environment. From a program's point of view, we talk about an escape condition. This can be caused by an escape key or the program itself.

By default, the key that causes an escape condition is Escape. RISC OS can be configured so that the escape key is any key on the keyboard.

When an escape condition occurs, RISC OS will call the escape handler of the program or the language environment. See the description of handlers in the section entitled *Handlers* on page 1-282. The escape handler or running program should then clear the escape condition and act in an appropriate way. Note that it is perfectly valid for a program to ignore an escape condition as long as it is cleared.

The escape event can also be enabled. This is called in place of the escape handler. (See the chapter entitled *Events* on page 1-137.)

Serial port

A character which comes into the serial port interrupts the computer. It is then placed into the serial input buffer, if it is enabled. RISC OS can be configured so that serial input is ignored.

The computer can be set up so that input coming in from the serial port is treated exactly as if it had come from the keyboard. This means that the escape character and function key codes will be recognised.

If characters come in the serial port too quickly to be processed, then the serial input buffer would become full. After this point, data would be lost. To solve this problem, the serial driver will notify the sender to stop transmitting before it gets full. From a program's point of view, this all happens invisibly.

Calls that are specific to the serial port, whether they refer to input or output (eg those to set the baud rate, or to explicitly send/receive a character from/to the serial port), are gathered together in the chapter entitled *Serial device* on page 3-419.

*Exec

Exec is the opposite of spooling, which is used in character output. Exec makes a file the current input stream. Keyboard and serial input is ignored.

A SWI is provided to allow the Exec file to be changed or stopped under program control.

Technical Details

Events

There are a number of events associated with the character input system. In particular:

- input buffer has become full
- character placed in input buffer
- a key has been pressed/released
- serial error has occurred
- escape condition detected

See the chapter entitled *Events* on page 1-137 for more details of these events.

Streams

OS_ReadC is the core of the input stream system. It is called by many SWIs and it uses one of the three streams as an input source. The stream that it uses can be controlled by OS_Byte 2 for keyboard and serial port. To use the third stream, the file, then *Exec or OS_Byte 198 can be used. OS_Byte 177 can be used to read the setting of the last OS_Byte 2.

OS_ReadC is also responsible for handling cursor-editing during input.

OS_ReadLine

OS_ReadLine, and its obsolete equivalent OS_Word 0, will read a line of input from the current input stream. It copes with the deleting of characters or the whole line. Thus, a single call which returns a simple string to the program allows the user much flexibility.

Keyboard

When a key is pressed (or released), a code unique to that key is transmitted to the computer through the keyboard connector cable. This code is read into some hardware, which causes an interrupt to occur. The keyboard driver responds to this interrupt by reading the keycode, and passing it on to the keyboard handler for further processing.

At this stage, a key press/release event may be generated, which you can handle as required. Also, at this level mouse button presses look exactly the same as any other key press. It is only when the mouse button presses reach the keyboard handler that they are recognised as such, and RISC OS is informed that the mouse button state has changed.

Keyboard buffer

The keyboard buffer in RISC OS is 255 characters long. It is often termed a type-ahead buffer, as it enables the user to type commands ahead of the program being ready for them.

Disabling buffering

OS_Byte 201 will stop the keyboard handler from putting any characters it gets into the keyboard buffer. This means that most keyboard reading calls will not work. Where this function is useful is if you want a program to insert codes directly into the buffer without any of the user's key strokes appearing in the middle of them.

Keyboard status

If the key pressed (or released) is one of the shifting keys (Shift, Ctrl or Alt) or one of the locking keys (Caps Lock, Num Lock or Scroll Lock) is pressed, then the key handler just makes a note of this fact by updating its status information. Normally this doesn't cause any character to be inserted into the keyboard buffer; although the Alt key can in combination with the numeric keypad – see *Table D: Character sets* on page 6-491.

OS_Byte 202 allows reading and writing of the keyboard status byte. This is a bitfield that represents the state of Shift, Ctrl, Alt and all the Lock keys. If it is written and any of the Lock keys with LEDs are changed, then this will not be reflected in the LEDs. OS_Byte 118 must be called to do this.

The next time a key goes down or up, then the Shift, Alt and Ctrl states will reflect their real position and the LEDs will be updated to their current status.

The Caps Lock key state can be set up using *Configure Caps, NoCaps and ShCaps.

Scanning keys

Scanning refers to being able to get the low level key codes without the buffering and interpretation that is placed on keys by the higher level routines. The internal key number returned is not the code that the keyboard itself sends the computer. This is translated to a standard internal key number that maintains compatibility with BBC/Master series keyboard codes.

There are three OS_Bytes that can scan the keyboard. OS_Byte 121 can scan a particular key or a range or keys. Like this call, OS_Byte 122 can scan a fixed range of keys, all but the Shift, Alt, Ctrl and mouse keys. OS_Byte 129 can scan a particular key, like OS_Byte 121. It can also read a key with a time limit. This is discussed later.

Key handler

The character stored in the keyboard buffer is derived from a table in the key handler, which maps keycodes into buffer codes, using the state of the various shifting and locking keys to alter the character if appropriate. In addition, the key-press is recorded in a 'last key pressed' location. This is to enable auto-repeating keys to be implemented, as described below.

For the standard keys, eg the letters, digits, punctuation marks etc, the buffer code is the ASCII code of the symbol. Thus when the code comes to be removed from the keyboard buffer (by OS_ReadC, for example), it is returned directly to the user. The other keys, such as the function keys and cursor keys, are entered as top-bit set characters, in the range &80 - &FF.

Custom key handler

The SWI OS_InstallKeyHandler allows replacing the module that decodes key numbers into ASCII. It is outside the scope of this manual to discuss this procedure in depth.

Read with time limit

OS_Byte 129 supports two operations, one of which, low level keyboard scanning, was discussed in the earlier section on scanning keys.

The other allows reading a character from the keyboard buffer within a time limit. This is useful in cases where a program waits for a response for a time, and if none is entered, continues. It can be used in a situation where the keyboard buffer needs to be checked periodically, but the program doesn't wish to be trapped waiting in OS_ReadC for a character to be entered. To achieve this, this call would be used with a very brief waiting time, so if no characters are available in the buffer, then the program can continue.

Tab key

OS_Byte 219 reads or modifies the code inserted into the keyboard buffer when the Tab key is pressed (the default is 9). If the value specified is in the range &80 to &FF, then the value to be inserted is modified by the state of the Shift and Ctrl keys, in a similar fashion to the function keys.

Auto-repeat

The auto-repeat of keys has two aspects. The delay before the key starts repeating and the rate of repeating. The delay can be read and changed with OS_Byte 196, or changed with OS_Byte 11. The rate can be read and changed with OS_Byte 197, or changed with OS_Byte 12. Both are adjustable from 1 to 255 centiseconds. Auto-repeat can also be disabled.

You can use OS_Byte 120 to lock auto-repeat until the key(s) currently depressed are released. An example of use would be where one place of input has changed to another and the program doesn't want any characters from one place auto-repeating and confusing the next.

The delay and rate can be set up using *Configure Delay and Repeat, which use the same parameter as the appropriate OS_Bytes.

Arrow and Copy keys

In a default system, these keys are used for on-screen editing. The arrows move a cursor and Copy copies the character that it is on to the second cursor.

OS_Byte 237 allows reading and changing how cursor keys are interpreted. As well as the default editing state, they can be in two other modes. In one, the keys return characters in the range 135 to 139. In the other, they act as function keys, and can be treated as all the other function keys.

OS_Byte 4 also allows changing this state.

Numeric keypad

There is a base value for the numeric keypad. A key on the numeric keypad adds an offset to this to get the character that is placed in the keyboard buffer. The offset of each key is such that the default base value of 48 will give each key the ASCII value of the character on the key.

This base value can be changed with OS_Byte 238. See the documentation on this call for details of the offsets of each key.

Shift and Ctrl can alter the value returned from the keypad. By default, this feature is disabled, but you can enable it with OS_Byte 254.

Interpreting characters &80 - &FF

When referring to function keys, we are talking about two separate things. There are the keys, many discussed earlier, that generate buffer codes in the range &80 to &FF. Then there is the interpretation placed upon these buffer codes by RISC OS as it reads them from the buffer.

Interpreting these keys as function keys is only one way of using them. OS_Bytes 221 - 228 allow control over how buffer codes from &80 to &FF are interpreted by RISC OS. Each OS_Byte handles a group of 16 characters. Each group can be configured so that its characters are:

- interpreted as function keys
- preceded by a NULL (ASCII 0)
- offset by any number from 3 - &FF
- discarded

Function keys

If a character is read from the keyboard buffer and is in a group that is configured as function keys, then a special action is taken by the keyboard handler. First of all, it looks up the value of the KeyS*n* system variable which corresponds to the function key. The function key number is the lower nibble of the character. Thus, if the character is &81, the variable read is KeyS1.

The variable refers to a string, which is copied into the function key buffer. If the string was a null string (the function key wasn't set), then RISC OS continues, removing the next character from the input buffer.

Otherwise, the first character is removed from the function key buffer and returned to the calling program. Characters read from this buffer are returned without interpretation in any way.

Subsequent calls to OS_ReadC and OS_Byte 129 spot that a function key is being read, and remove characters from the function key buffer instead of looking in the input buffer. This continues until the last character has been read from the buffer. Input then reverts to the normal input buffer.

OS_Byte 216 is used to see how much of a function key string remains to be read from the function key buffer. It can also change this value, to terminate for instance, but must be used with care.

Setting and clearing

To set a function key, a number of commands can be called:

- *Key *n string*
- *Set KeyS*n string*
- *SetMacro KeyS*n expression*. This is passed through OS_GSTrans when it is copied to the function key buffer. This is interesting because it means that the string generated by a function key can change every time it is used.

To reset one or more function keys, there is also a variety of commands that can be used:

• *Key *n*	will reset function key *n*
• *Unset KeyS*n*	will also reset function key *n*
• *Unset KeyS*	will reset all function keys
• OS_Byte 18	will also reset all function keys

Reset, Break and Escape

Reset

When you press the Reset button, then the RISC OS ROM is paged into the bottom of memory and performs certain housekeeping actions. It then pages itself out and restarts the system.

A soft reset distinguishes itself from a hard reset in a matter of degree. A hard reset will initialise far more things in the system. A soft reset, for instance, will not change the settings for PrinterType and the printer ignore character. It will reset vectors that have been claimed however.

OS_Byte 200 sets whether a reset will act as described above or will cause a complete memory clear. This makes it a power-on reset. If this is used, then all things kept in memory will be lost and *Configure settings restored. This command should be used with discretion because of its powerful effects.

OS_Byte 253 can be used to see what kind of reset the last one was.

Break

Break is configurable with OS_Byte 247. This sets how Break, Shift Break, Ctrl Break and Ctrl Shift Break act. They can each be set to cause a reset or an escape or have no effect. A reset caused by the break key does not page the ROM into the bottom of memory (as one caused by the Reset button does); instead, it just jumps to the correct location in the ROM.

Escape

The diagram below illustrates how all the calls in the escape system work together. A description of this interaction follows the diagram.

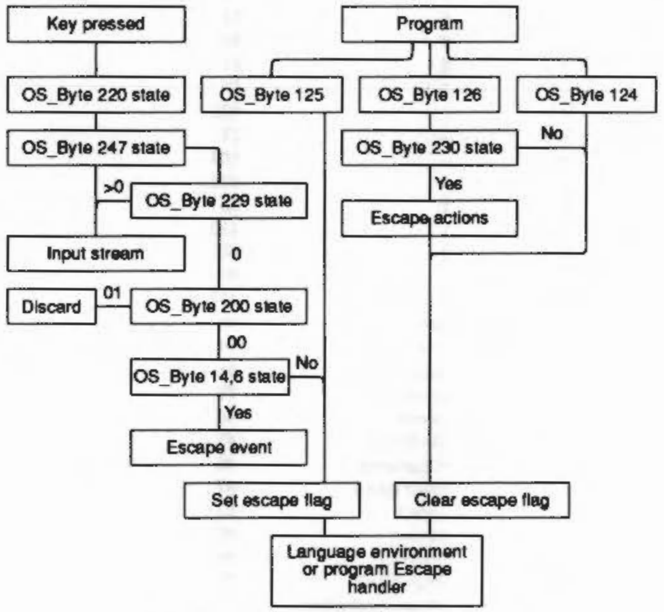


Figure 23.1 Interaction of calls in the escape system

Causing escape

An escape condition can be caused by a key or under program control. By default, the escape key is Escape. OS_Byte 220 can read or alter which key will cause an escape condition. OS_Byte 247 can alter the Break key (or Shift and Ctrl modifiers of it) so that it causes an escape condition. Thus, it is possible to have two escape keys on the keyboard, and this is indeed the default state.

Under program control, OS_Byte 125 can force an escape condition to occur. Note that it will not generate an event, but the escape handler is called.

OS_ReadEscapeState can check whether an escape condition has occurred. It can be called at any time, even from within interrupts.

Disabling escape

OS_Byte 229 controls recognition of this escape character. It can disable the effect of the escape character and allow it to pass through the input stream unaltered. OS_Byte 200 can disable all escape conditions apart from those caused by OS_Byte 125. In this case, any escape characters would be discarded.

OS_Byte 14,6, which is described in the chapter entitled *An introduction to* RISC OS controls whether the escape event is enabled or not. If the escape event is enabled, then it will be called and not the escape handler.

After an escape

OS_Byte 126 will acknowledge an escape condition and call the escape handler to clear up. OS_Byte 124 will clear an escape condition without calling the escape handler.

OS_Byte 230 controls whether the normal effects of an escape occur or not when it is acknowledged. These include flushing buffers, closing the Exec file, terminating any sounds and so on.

Serial device

The serial device is provided as a DeviceFS (*Device Filing System*) device. For full details, see the chapter entitled *DeviceFS* on page 3-401, and the chapter entitled *Serial device* on page 3-419. The latter chapter also contains all calls that are specific to the serial port, whether they refer to input or output (eg those to set the baud rate, or to explicitly send/receive a character from/to the serial port).

*Exec

There are two ways of causing a file to be made the input stream. The simplest is to use *Exec, which will open the specified file and attach it as the input stream. For more control, OS_Byte 198 does what *Exec does and can also terminate the Exec stream at any time or change to another file.

Internal key numbers

Here is a list of the BBC/Master compatible internal key numbers in order of key category and in numerical order.

By category

Key	**Internal key number**
Print (F0)	32
F1	113
F2	114
F3	115
F4	20
F5	116
F6	117
F7	22
F8	118
F9	119
10	30
F11	28
F12	29
A	65
B	100
C	82
D	50
E	34
F	67
G	83
H	84
I	37
J	69
K	70
L	86
M	101
N	85
O	54
P	55
Q	16
R	51
S	81
T	35
U	53
V	99
W	33
X	66
Y	68
Z	97
0	39
1	48
2	49
3	17
4	18
5	19
6	52
7	36
8	21
9	38
,	102
–	23
.	103
/	104
[	56
\	120
]	88
;	87
Escape	112
Tab	96
Caps Lock	64
Scroll Lock	31
Num Lock	77
Break	44
Back tick/~	45
£/currency	46
Back space	47
Insert	61
Home	62
Page Up	63
Page Down	78
Single or double quotes	79
Shift (either or both)	0
Ctrl (either or both)	1
Alt (either or both)	2
Shift (left-hand)	3
Ctrl (left-hand)	4
Alt (left-hand)	5
Shift (right-hand)	6
Ctrl (right-hand)	7
Alt (right-hand)	8
Space Bar	98
Delete	89

Return	73
Copy	105
Up arrow	57
Right arrow	121
Left arrow	25
Down arrow	41
keypad 0	106
keypad 1	107
keypad 2	124
keypad 3	108
keypad 4	122
keypad 5	123
keypad 6	26
keypad 7	27
keypad 8	42
keypad 9	43
keypad +	58
keypad –	59
keypad .	76
keypad /	74
keypad #	90
keypad *	91
keypad Enter	60
Left mouse button	9
Centre mouse button	10
Right mouse button	11
(extra)	94

Some international keyboards have an extra key to the right of the left hand shift key. This is the extra key 94

In order

Key	**Internal key number**	
Shift (either or both)	0	
Ctrl (either or both)	1	
Alt (either or both)	2	
Shift (left-hand)	3	
Ctrl (left-hand)	4	
Alt (left-hand)	5	
Shift (right-hand)	6	
Ctrl (right-hand)	7	
Alt (right-hand)	8	
Left mouse button	9	
Centre mouse button	10	
Right mouse button	11	
Q	16	
3	17	
4	18	
5	19	
F4	20	
8	21	
F7	22	
–	23	
^	24	(synonym, kept for Master compatibility)
Left arrow	25	
keypad 6	26	
keypad 7	27	
F11	28	
F12	29	
F10	30	
Scroll Lock	31	
Print (F0)	32	
W	33	
E	34	
T	35	
7	36	
I	37	
9	38	
0	39	
_	40	(synonym, kept for Master compatibility)
Down arrow	41	
keypad 8	42	
keypad 9	43	
Break	44	(but see OS_Byte 247 – it may cause a reset).
Back tick/–	45	
£/currency	46	
Back space	47	
1	48	
2	49	
D	50	
R	51	
6	52	
U	53	
O	54	
P	55	
[	56	

Up arrow	57	
keypad +	58	
keypad –	59	
keypad Enter	60	
Insert	61	
Home	62	
Page Up	63	
Caps Lock	64	
A	65	
X	66	
F	67	
Y	68	
J	69	
K	70	
@	71	(synonym, kept for Master compatibility)
:	72	(synonym, kept for Master compatibility)
Return	73	
keypad /	74	
keypad .	76	
Num Lock	77	
Page Down	78	
Single or double quotes	79	
S	81	
C	82	
G	83	
H	84	
N	85	
L	86	
;	87	
]	88	
Delete	89	
keypad #	90	
keypad *	91	
=	93	
(extra)	94	

(Some international keyboards have an extra key to the right of the left hand shift key. This is the extra key 94.)

Tab	96
Z	97
Space Bar	98
V	99
B	100

M	101
,	102
.	103
/	104
Copy	105
keypad 0	106
keypad 1	107
keypad 3	108
Escape	112
F1	113
F2	114
F3	115
F5	116
F6	117
F8	118
F9	119
\	120
Right arrow	121
keypad 4	122
keypad 5	123
keypad 2	124

Service Calls

Service_KeyHandler (Service Call &44)

Keyboard handler

On entry

R1 = &44 (reason code)
R2 = keyboard ID

On exit

R1 preserved to pass on (don't claim)
R2 preserved

Use

This call is made on reset, when the OS has established which type of keyboard is present, and after an OS_InstallKeyHandler SWI. It is for the information of keyboard handler modules which need to know what sort of keyboard is present; it should not be claimed.

The Archimedes 300, 400 and 500 series, the A3000, the R140 and the R200 series all have a keyboard ID of 1.

SWI Calls

OS_ReadC (SWI &04)

Read a character from the input stream

On entry

—

On exit

if C flag = 0 then R0 = ASCII code
if C flag = 1 then R0 = error type: &1B in R0 means an escape

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call will read a character from the input stream. OS_Byte 2 can be used to change the selection of the current input stream.

Cursor key presses go into the buffer. When OS_ReadC reads a cursor key code from the buffer it handles the cursor editing for you, assuming the cursor keys are set up to do cursor editing. That is, if one of the arrow keys is pressed, cursor edit mode is entered, indicated by the presence of two cursors on the screen. You can copy characters from underneath the input cursor by pressing Copy. The character read is returned from the routine as if you had typed it explicitly.

Cursor editing only applies if enabled (see OS_Byte 4) and is cancelled when ASCII 13 is sent to the VDU driver.

Related SWIs

OS_Byte 2 (SWI &06), OS_ReadLine (SWI &0E)

Related vectors

RdchV

OS_Byte 2 (SWI &06)

Specify input stream

On entry

R0 = 2
R1 = stream selection (0, 1 or 2)

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call selects the device from which all subsequent input is taken by OS_ReadC. This is determined by the value of R1 passed as follows:

- 0 for keyboard input with serial input buffer disabled
- 1 for serial input
- 2 for keyboard input with serial input buffer enabled

The difference between the 0 and 2 values is that the latter allows characters to be received into the serial input buffer under interrupts at the same time as the keyboard is being used as the primary input. If the input stream is subsequently switched to the serial device, then those characters can then be read.

Note that the value returned in R1 from this call is:

- 0 when input was from the keyboard
- 1 when input was from the serial port

The state of this variable can be read by OS_Byte 177.

Related SWIs

OS_Byte 177 (SWI &06)

Related vectors

ByteV

OS_Byte 4 (SWI &06)

Cursor key status

On entry

R0 = 4
R1 = new state

On exit

R0 preserved
R1 = state before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call alters the effect of the four arrow keys and the Copy key. The value of R1 determines their state:

0 Enables cursor editing. This is the default state.

1 Disables cursor editing. When pressed, the keys return the following ASCII values:

Key	**Value**
Copy	135
Left arrow	136
Right arrow	137
Down arrow	138
Up arrow	139

2 Cursor keys act as function keys. The function key numbers assigned are:

Key	Function key number
Copy	11
Left arrow	12
Right arrow	13
Down arrow	14
Up arrow	15

OS_Byte 237 may be used to write and read this state.

Related SWIs

OS_Byte 237 (SWI &06)

Related vectors

ByteV

OS_Byte 11 (SWI &06)

Write keyboard auto-repeat delay

On entry

R0 = 11
R1 = delay period in centiseconds

On exit

R0 preserved
R1 = previous delay period
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

You must hold down each key on the keyboard for a number of centiseconds before it begins to autorepeat. This call enables you to change the initial delay from the default set by *Configure Delay.

If the delay period is zero, then auto-repeat is disabled.

This variable may also be read and set using OS_Byte 196.

Related SWIs

OS_Byte 12 (SWI &06), OS_Byte 196 (SWI &06)

Related vectors

ByteV

OS_Byte 12 (SWI &06)

Write keyboard auto-repeat rate

On entry

R0 = 12
R1 = repeat rate in centiseconds (unless R1 = 0)

On exit

R0 preserved
R1 = previous repeat rate
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

After the auto-repeat delay specified by OS_Byte 11, each key will repeat until released at the rate passed to this call. This call enables you to change the initial rate from the default set by *Configure Repeat. One particular use of this is to speed up cursor editing.

If the rate is zero, then the auto-repeat and delay values are reset to their configured settings.

This variable may also be read and set using OS_Byte 197.

Related SWIs

None

Related vectors

ByteV

OS_Byte 18 (SWI &06)

Reset function key definitions

On entry

R0 = 18

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call removes all of the Key$*n* variables, which contain the function key definitions. It also cancels any key string currently being read.

You can also clear individual strings by *Key *n*, or all of them by *Unset Key$*. Neither of these commands cancel the current key expansion, though.

Related SWIs

None

Related vectors

ByteV

OS_Byte 118 (SWI &06)

Reflect keyboard status in LEDs

On entry

R0 = 118

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The settings of Caps Lock, Scroll Lock and Num Lock are held in a location referred to as the keyboard status byte. See OS_Byte 202 on page 2-390 for detail of this.

Under normal circumstances they are shown by the keyboard LEDs which are set into the keycaps. However, the keyboard status byte is written to using OS_Byte 202, then the LEDs will not update. This call ensures that the current contents of the keyboard status byte are reflected in the LEDs.

Related SWIs

OS_Byte 202 (SWI &06)

Related vectors

ByteV

OS_Byte 120 (SWI &06)

Temporarily lock auto-repeat

On entry

R0 = 120
R1 = 0
R2 = 0

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call locks the auto-repeat mechanism for the duration of a key being down. This is useful when input is followed by further input, but no auto-repeat is desired from the key that may still be down.

This call is kept for compatibility with the BBC/Master series.

Related SWIs

None

Related vectors

ByteV

OS_Byte 121 (SWI &06)

Keyboard scan

On entry

R0 = 121
R1 = key(s) to be detected

On exit

R0 preserved
R1 = if/which key has been detected
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call allows checking the keyboard to see whether a particular key or a range of keys is being pressed. It uses the internal key number (see the section entitled *Internal key numbers* on page 2-349 for a complete list).

Single key

To check for a single key, R1 must contain the internal key number exclusive ORd with &80 (R1 EOR &80). The value returned in R1 will be &FF if that key is currently down and zero if it is not.

Key range

To check for a range of key values, it is possible to set the 'low tide' mark. That is, no internal key number below the value in R1 on entry will be recognised. Since Shift, Ctrl, Alt and the mouse keys are at the bottom then this is very convenient.

The value returned in R1 will be the internal key number if a key is currently down or &FF if no key is down.

Related SWIs

OS_Byte 122 (SWI &06)

Related vectors

ByteV

OS_Byte 122 (SWI &06)

Keyboard scan (other than Shift, Ctrl, Alt and mouse keys)

On entry

R0 = 122

On exit

R0 preserved
R1 = internal key number of key, or &FF if none
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call allows checking the keyboard to see whether any key is being pressed. It uses the internal key number (see the section entitled *Internal key numbers* on page 2-349 for a complete list). All key numbers below 16 are ignored. This excludes all Shift, Ctrl, Alt and mouse keys. It is equivalent to calling OS_Byte 121 with R1 = 16.

Related SWIs

OS_Byte 121 (SWI &06)

Related vectors

ByteV

OS_Byte 124 (SWI &06)

Clear escape condition

On entry

R0 = 124

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call clears any escape condition by calling the escape handler with R11 = 0, and then returns.

Related SWIs

OS_Byte 125 (SWI &06), OS_Byte 126 (SWI &06)

Related vectors

ByteV

OS_Byte 125 (SWI &06)

Set escape condition

On entry

R0 = 125

On exit

R0 preserved
R1, R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call is used to set the escape flag and call the escape handler. An escape event is not generated.

Related SWIs

OS_Byte 124 (SWI &06), OS_Byte 126 (SWI &06)

Related vectors

ByteV

OS_Byte 126 (SWI &06)

Acknowledge escape condition

On entry

R0 = 126

On exit

R0 preserved
R1 = indicates if the escape condition has been cleared
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call attempts to clear an escape condition if one exists. It may or may not need to perform various actions to tidy up after the escape condition depending on whether the escape condition side effects (see OS_Byte 230) have been enabled or not.

The escape handler is called to indicate clearing of the escape condition.

The value returned in R1 indicates whether or not the escape condition has been cleared. &FF indicates success, while zero means that there wasn't an escape condition to clear.

Related SWIs

OS_Byte 124 (SWI &06), OS_Byte 125 (SWI &06), OS_Byte 230 (SWI &06)

Related vectors

ByteV

OS_Byte 129 (SWI &06)

Read keyboard for information

On entry

R0 = 129

To read a key within a time limit:
R1 = time limit low byte
R2 = time limit high byte (in range &00 - &7F)

To read the OS version identifier:
R1 = 0
R2 = &FF

To scan the keyboard for a range of keys:
R1 = lowest internal key number EOR &7F (ie a value of &01 - &7F)
R2 = &FF

To scan the keyboard for a particular key:
R1 = internal key number EOR &FF (ie a value of &80 - &FF)
R2 = &FF

On exit

R0 preserved

If reading a key within a time limit:
R1 = ASCII code if character read, else undefined
R2 = &00 if character read
&1B if an escape condition exists
&FF if timeout

If reading the OS version identifier:
R1 = &A0 (Arthur 1.20), &A1 (RISC OS 2.00), &A2 (RISC OS 2.01) or &A4 (RISC OS 3.00)
R2 = &00

If scanning the keyboard for a range of keys:
R1 = internal key number or &FF if none pressed
R2 is corrupted

If scanning the keyboard for a particular key:
R1 = &FF if the required key was pressed, 0 otherwise
R2 = &FF if the required key was pressed, 0 otherwise

Interrupts

Interrupt status:

- Enabled when reading a key within a time limit
- Not altered for remaining three operations

Fast interrupts are enabled for all operations

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This OS_Byte is four separate operations in one:

- read an ASCII key value read from the keyboard with a timeout
- read the OS version identifier
- scan the keyboard for a range of keys
- scan the keyboard for a particular key.

Read key with time limit

In this operation, RISC OS waits up to a specified time for a key to be pressed, if there are none in the keyboard buffer.

The time limit is set according to the following calculation:

R1+(R2*256) centiseconds

The upper limit is 32767 centiseconds. To indicate the time of (n) centiseconds, then:

R1 = n MOD &100
R2 = n DIV &100

If an escape condition is detected during this operation it should be acknowledged by the application using OS_Byte 126, or cleared using OS_Byte 124.

While RISC OS is waiting for a keyboard character during one of these calls, it also deals with cursor key presses. That is, if one of the arrow keys is pressed, cursor edit mode is entered, indicated by the presence of two cursors on the screen. You can copy characters from underneath the input cursor by pressing Copy. The character read is returned from the routine as if you had typed it explicitly. Cursor editing is cancelled when Return (ASCII 13) is sent to the VDU driver. Cursor editing can be disabled with OS_Byte 4.

Read the OS version identifier

If R2=&FF and R1=0, then the OS version identifier is read.

Scan for a range of characters

If R2=&FF and R1 is in the range &1 to &7F, then the keyboard is scanned for any keys that are being pressed, which have an internal key number greater than or equal to R1 EOR &7F. If found, the internal key number is returned. If no key is found, then &FF is returned.

Scan for a particular key

If R2=&FF and R1 is in the range &80 to &7F, then the keyboard is scanned for a particular key with internal key number equal to R1 EOR &FF.

In BBC/Master series computers, the internal key numbers are the same as the keyboard scan numbers; but the two differ for other Acorn computers.

A list of all internal key numbers can be found in the section entitled *Internal key numbers* on page 2-349.

Related SWIs

None

Related vectors

ByteV

OS_Byte 177 (SWI &06)

Read input stream selection

On entry

R0 = 177
R1 = 0
R2 = 255

On exit

R0 preserved
R1 = value of stream selection
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This returns the number of the buffer from which character input gets characters:

- 0 when input was from the keyboard
- 1 when input was from the serial port

You must not alter this number with this call by using other values in R1 and R2.

Related SWIs

OS_Byte 2 (SWI &06)

Related vectors

ByteV

OS_Byte 178 (SWI &06)

Read/write keyboard semaphore

On entry

R0 = 178
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

This call is obsolete and should not be used.

Related SWIs

None

Related vectors

ByteV

OS_Byte 196 (SWI &06)

Read/write keyboard auto-repeat delay

On entry

R0 = 196
R1 = 0 to read or new delay to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 = keyboard auto-repeat rate (see OS_Byte 197)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The delay stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((delay AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call can read and set the keyboard auto-repeat delay value. OS_Byte 11 can also write this variable, and has more information about it.

Related SWIs

OS_Byte 11 (SWI &06), OS_Byte 12 (SWI &06)

Related vectors

ByteV

OS_Byte 197 (SWI &06)

Read/write keyboard auto-repeat rate

On entry

R0 = 197
R1 = 0 to read or new rate to write
R2 = 255 to read or 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 rate stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((rate AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call can read and set the keyboard auto-repeat rate value. OS_Byte 12 can also write this variable, and has more information about it. Note the difference between *FX 12,0 (which sets the auto-repeat rate and delay to their configured values) and *FX 197,0 (which sets the auto-repeat rate to zero).

Related SWIs

OS_Byte 11 (SWI &06), OS_Byte 12 (SWI &06)

Related vectors

ByteV

OS_Byte 198 (SWI &06)

Read/write *Exec file handle

On entry

R0 = 198
R1 = 0 to read or new handle to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The handle stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((handle AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This command can be used to read or write the location that holds the Exec file handle.

If reading, it can tell whether an Exec file is the current input stream or not. Any non-zero number is a handle and hence the input stream.

If writing a handle over a zero, then it causes the same effect as a *Exec command.

If writing over a Exec file handle, the current Exec file will be switched off. This handle, which is returned, should then be properly closed after use. If you write a new handle value in its place, then this has the effect of switching input in mid-stream. If you write a zero in this case, then it will have terminate the current input stream.

In both these cases care must be taken not to cause the Exec file to stop at an inconvenient point.

If you are writing a file handle, the new file must be open for input or update, otherwise a Channel error occurs. If an attempt is made to use a write-only file for the *Exec file, a Not open for reading error is given.

Related SWIs

None

Related vectors

ByteV

OS_Byte 200 (SWI &06)

Read/write Break and Escape effect

On entry

R0 = 200
R1 = 0 to read or new state to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = state before being overwritten
R2 = keyboard disable flag (see OS_Byte 201)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The state stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((state AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call can read or change the effects of a reset (including resets caused by Break) and of Escape.

The bottom two bits of R1 have the following significance:

Bit	Value	Effect
0	0	Normal escape action
	1	Escape disabled unless caused by OS_Byte 125
1	0	Normal reset action
	1	Power on reset (only if bits 2 - 7 of R1 are zero) This means a value of 2_0000001x causes a memory clear.

Related SWIs

None

Related vectors

ByteV

OS_Byte 201 (SWI &06)

Read/write keyboard disable flag

On entry

R0 = 201
R1 = 0 to read or new flag to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = flag before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The flag stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((flag AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call allows you to read and change the keyboard state (ie whether the keyboard is enabled or disabled). When it is enabled, all keys are read as normal. When it is disabled, the keyboard interrupt service routine does not place these keys into the keyboard buffer.

A value of zero will enable keyboard input, while any non-zero value will disable it.

Related SWIs

None

Related vectors

ByteV

OS_Byte 202 (SWI &06)

Read/write keyboard status byte

On entry

R0 = 202
R1 = 0 to read or new status to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = status before being overwritten
R2 = serial input buffer space (see OS_Byte 203)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The status stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((status AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

The keyboard status byte holds information on the current status of the keyboard, such as the setting of Caps Lock. This call enables you to read and change these settings.

The bit pattern in R1 determines the settings. In this table, the State column has on and off in it. *On* means a LED is lit or a key is pressed, and *off* means the opposite. Take careful note of the state, because they are not all in the same order:

Bit	Value	State	Meaning
0	—	—	Reserved for use by keyboard handler: must be preserved when writing
1	0	off	Scroll Lock
	1	on	
2	0	on	Num Lock
	1	off	
3	0	off	Shift
	1	on	
4	0	on	Caps Lock
	1	off	
5			Normally set
6	0	off	Ctrl
	1	on	
7	0	off	Shift Enable
	1	on	

If Caps Lock is on, then Shift will have no effect. If Shift Enable and Caps Lock are on, then Shift will get lower case. You can enter this state from the keyboard by holding Shift down and pressing Caps Lock.

This call does not update the LEDs. The next key down or up event will update them, or you can call OS_Byte 118.

Related SWIs

OS_Byte 118 (SWI &06)

Related vectors

ByteV

OS_Byte 216 (SWI &06)

Read/write length of function key string

On entry

R0 = 216
R1 = 0 to read or new length to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = length before being overwritten
R2 = paged mode line count (see OS_Byte 217)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The length stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((length AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call reads and changes the count of characters left in the currently active function key definition. An active function key is one that is being read by OS_ReadC instead of the current input stream.

If the length is zero, then no function key string is being read. A zero length must never be changed with this call.

A non-zero value shows that a function key string is active. Setting it to zero effectively cancels that function key from that point. Changing it to any non-zero value will have an indeterminate effect.

Related SWIs

None

Related vectors

ByteV

OS_Byte 219 (SWI &06)

Read/write Tab key value

On entry

R0 = 219
R1 = 0 to read or new value to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

OS_Byte 219 reads or modifies the code inserted into the keyboard buffer when the Tab key is pressed (the default is 9). If the value specified is in the range &80 to &FF, then the value to be inserted is modified by the state of the Shift and Ctrl keys as follows:

- Shift exclusive ORs the value with &10
- Ctrl exclusive ORs the value with &20

The value inserted will be interpreted by OS_ReadC in the normal way. For example, if the value specified is &82, then the Tab key behaves in an identical way to the function key F2.

Related SWIs

None

Related vectors

ByteV

OS_Byte 220 (SWI &06)

Read/write escape character

On entry

R0 = 220
R1 = 0 to read or new value to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call can read and change the character that will cause an escape condition when it is read from the input stream. Escape (ASCII 27) is the default.

For example:

Value	Key that causes an escape condition
27	Escape
53	'5'
&81	F1
&A1	Ctrl F1

Related SWIs

None

Related vectors

ByteV

OS_Bytes 221 - 228 (SWI &06)

Read/write interpretation of buffer codes

On entry

R0 = 221 - 228
R1 = 0 to read or new value to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call provides a way of reading and changing how the codes from &80 to &FF are interpreted when read from the input buffer.

They are split into eight groups as follows:

OS_Byte	Range of buffer codes controlled
221	&C0 - &CF
222	&D0 - &DF
223	&E0 - &EF
224	&F0 - &FF
225	&80 - &8F
226	&90 - &9F
227	&A0 - &AF
228	&B0 - &BF

The list below shows the keys that can produce codes in these groups:

Key	Code	+Shift	+Ctrl	+Ctrl-Shift
Print	&80	&90	&A0	&B0
F1	&81	&91	&A1	&B1
F2	&82	&92	&A2	&B2
:	:	:	:	:
F9	&89	&99	&A9	&B9
Copy	&8B	&9B	&AB	&BB
←	&8C	&9C	&AC	&BC
→	&8D	&9D	&AD	&BD
↓	&8E	&9E	&AE	&BE
↑	&8F	&9F	&AF	&BF
Page Down	&9E	&8E	&BE	&AE
Page Up	&9F	&8F	&BF	&AF
F10	&CA	&DA	&EA	&FA
F11	&CB	&DB	&EB	&FB
F12	&CC	&DC	&EC	&FC
Insert	&CD	&DD	&ED	&FD

These SWIs only affect the codes generated by the Copy and arrow keys if they have been set up to act as function keys by calling OS_Byte 4 with R1 = 2. Normally this is not the case, and you should use OS_Byte 4 to control the action of these keys.

Also, when a reset occurs, the code &CA is inserted into the input buffer. This causes the key definition for function key 10 to be used for subsequent input if it is defined.

Some of these codes cannot be generated from the main keyboard, but must be produced via one of the following techniques:

- use these calls to generate them with keys
- re-base the numeric keypad with OS_Byte 238
- insert into the buffer with OS_Byte 138
- insert into the buffer with OS_Byte 153
- receive via the serial input port

The interpretation of these codes depends upon the value of R1 passed. This is the interpretation value. It determines what action will be taken with a code in the appropriate block:

Value	**Interpretation**
0	discard the code
1	generates the string assigned to function key (code MOD 16)
2	generates a NULL (ASCII 0) followed by the code
3 - &FF	acts as offset: ie (code MOD 16) + value

If any block has been set to interpretation value 2, then a Ctrl-@ (ASCII 0) will be passed as two zeros to differentiate it from a high code. This mode is used with software that can cope with the international character set in the range &A0 - &FF. It is recommended that the function keys return a NULL followed by the key code, so that they can be distinguished from actual ASCII characters in this range.

This is the default setting for each of the blocks:

Block	**Default**	**Interpretation**
&80 - &8F	1	function keys
&90 - &9F	&80	return (buffer code – &10)
&A0 - &AF	&90	return (buffer code – &10)
&B0 - &BF	0	discard
&C0 - &CF	1	function keys
&D0 - &DF	&D0	return buffer code unchanged
&E0 - &EF	&E0	return buffer code unchanged
&F0 - &FF	&F0	return buffer code unchanged

Related SWIs

OS_Byte 4 (SWI &06), OS_Byte 138 (SWI &06), OS_Byte 153 (SWI &06), OS_Byte 238 (SWI &06)

Related vectors

ByteV

OS_Byte 229 (SWI &06)

Read/write Escape key status

On entry

R0 = 229
R1 = 0 to read or new status to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = status before being overwritten
R2 = escape effects (see OS_Byte 230)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The status stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((status AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call allows you to enable or disable the generation of escape conditions, and to read the current setting. Escape conditions may be caused by pressing the current escape character or by the inserting it into the input buffer with OS_Byte 153.

If the value of R1 passed is zero, which is the default, then escape conditions are enabled. Any non-zero value will disable them. When they are disabled, the current escape character set by OS_Byte 220 will pass through the input stream unaltered.

OS_Byte 200 can also control the enabling of escape conditions.

Related SWIs

OS_Byte 153 (SWI &06), OS_Byte 200 (SWI &06), OS_Byte 220 (SWI &06)

Related vectors

ByteV

OS_Byte 230 (SWI &06)

Read/write escape effects

On entry

R0 = 230
R1 = 0 to read or new status to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = status before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The status stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((status AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

By default, the acknowledgement of an escape condition produces the following effects:

- Flushes all active buffers
- Closes any currently open *Exec file
- Clears the VDU queue
- Clears the VDU line count used in paged mode
- Terminates the sound being produced.

This call enables you to determine whether the escape effects are currently enabled or disabled, and to change the setting if required.

If the value of R1 passed is zero, which is the default, then escape effects are enabled. Any non-zero value will disable them.

Related SWIs

None

Related vectors

ByteV

OS_Byte 237 (SWI &06)

Read/write cursor key status

On entry

R0 = 237
R1 = 0 to read or new state to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 = numeric keypad interpretation (see OS_Byte 238)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The state stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((state AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This can read and modify the cursor key status. OS_Byte 4 can perform an identical write operation. See the description of that SWI in this chapter for details of the status.

Related SWIs

OS_Byte 4 (SWI &06)

Related vectors

ByteV

OS_Byte 238 (SWI &06)

Read/write numeric keypad interpretation

On entry

R0 = 238
R1 = 0 to read or new value to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call controls the character which is inserted into the input buffer when you press one of the keypad keys. The inserted character is derived from the sum of a base value (set by this call) and an offset, which depends on the key pressed. The inner (lighter) keys have two different offsets. The offset used depends on the state of Num Lock.

By default, the base number is 48: ie they generate codes which are displacements from 48 (ASCII '0').

This table shows the effect of the default settings on the keypad:

Key	Base Offset	Character Generated	Num Lock Offset	Character Generated
0	0	0	+157	
1	+1	1	+91	Copy
2	+2	2	+94	Down
3	+3	3	+110	Page Down
4	+4	4	+92	Left
5	+5	5	ignored	
6	+6	6	+93	Right
7	+7	7	-18	Home
8	+8	8	+95	Up
9	+9	9	+111	Page Up
.	-2	.	+79	Delete
/	-1	/	unchanged	
*	-6	*	unchanged	
#	-13	#	unchanged	
-	-3	-	unchanged	
+	-5	+	unchanged	
Enter	-35	Return	unchanged	

Unlike the function keys, you can set the numeric keypad base number to any value in the range 0 - 255. (If a generated code lies outside this range it is reduced MOD 256). If a character generated by the numeric keypad is in the range &80 to &8F, then it will act like a soft function key.

OS_Byte 254 controls how Shift and Ctrl act upon numeric keypad characters.

Related SWIs

OS_Byte 254 (SWI &06)

Related vectors

ByteV

OS_Byte 247 (SWI &06)

Read/write Break key actions

On entry

R0 = 247
R1 = 0 to read or new value to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call reads and changes the result of pressing Break. The value byte alters Break and modifiers of it as follows:

Bits	Key Combination
0,1	Break
2,3	Shift Break
4,5	Ctrl Break
6,7	Ctrl Shift Break

Each two bit number may take on one of these values:

Value	Effect
00	Act as Reset
01	Act as escape key
10	No effect
11	Undefined

The default is 2_00000001, so Break causes an escape condition.

Related SWIs

None

Related vectors

ByteV

OS_Byte 253 (SWI &06)

Read last reset type

On entry

R0 = 253
R1 = 0
R2 = 255

On exit

R0 preserved
R1 = break type
R2 = effect of Shift on keypad (see OS_Byte 254)

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the type of the last reset performed in R1:

Value	Reset type
0	Soft reset
1	Power-on reset
2	Hard reset

Related SWIs

None

Related vectors

ByteV

OS_Byte 254 (SWI &06)

Read/write effect of Shift and Ctrl on numeric keypad

On entry

R0 = 254
R1 = 0 to read or new value to write
R2 = 255 to read or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This call allows you to enable or disable the effect of Shift and Ctrl on the numeric keypad or to read the current state. These keys may modify the code just before it is inserted into the input buffer.

If the value of R1 passed is zero, then Shift and Ctrl are enabled. Any non-zero value will disable them; this is the default.

If they are enabled then the following actions occur depending on the value generated by a key:

- if the value ≥ &80:
  Shift exclusive ORs the value with &10
  Ctrl exclusive ORs the value with &20
- if the value < &80:
  Shift and Ctrl still have no effect

Related SWIs

None

Related vectors

ByteV

OS_Word 0 (SWI &07)

Read a line from input stream to memory

On entry

R0 = 0
R1 = pointer to parameter block

On exit

R0 preserved
R1 = preserved (and parameter block unaltered)
R2 = length of input line, not including the Return
the C flag is set if input is terminated by an escape condition

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call is equivalent to OS_ReadLine, but has the restriction that the parameter block must lie in the bottom 64k of memory. It is provided for compatibility with older Acorn operating systems.

The parameter block pointed to has the following structure:

Offset	Purpose	Equivalent in OS_ReadLine
0	LSB of buffer address	R0
1	MSB of buffer address	
2	size of buffer	R1
3	lowest ASCII code	R2
4	highest ASCII code	R3

Note that the parameter block must lie between &8000 and &FFFF in memory, never in &0000 to &7FFF, as this memory is reserved for RISC OS.

Related SWIs

OS_ReadLine (SWI &0E)

Related vectors

WordV

OS_ReadLine (SWI &0E)

Read a line from the input stream

On entry

R0 = pointer to buffer to hold the line (bits 0-29), and flags (bits 30-31)
bit 31 set ⇒ echo only those characters that set the buffer
bit 30 set ⇒ echo characters by echoing the character in R4
R1 = size of buffer
R2 = lowest ASCII value to pass
R3 = highest ASCII value to pass
R4 = character to echo if bit 30 of R0 is set

On exit

R0 corrupted
R1 = length of buffer read, not including Return.
R2, R3 corrupted
the C flag is set if input is terminated by an escape condition

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_ReadLine reads a line of text from the current input stream using OS_ReadC.

Input can be terminated in a number of ways:

- Return (ASCII 13). The length returned in R1 will not count the Return character, even though it is placed in the read buffer.
- Ctrl-J (ASCII 10 or linefeed). Acts much like the Return case above. Even the last character in the buffer is a Return, not a linefeed as you might expect.

- Escape condition. This can represent the escape key being pressed, but it can also be caused by other means, such as an OS_Byte 125.

With the exception of the above characters, and three more noted below, all characters received by OS_ReadLine will be echoed to OS_WriteC. Characters in the range R2 to R3 are also written into the read buffer that R0 points to on entry. These are the three characters that have a special function so are not placed in the buffer:

- Delete (ASCII 127) or Backspace (ASCII 8) act in the same way. They cause a Delete to be sent to OS_WriteC and the character last written into the buffer is removed.
- Ctrl-U (ASCII 21) deletes all the characters placed in the buffer and sends that many Deletes to OS_WriteC, effectively erasing the line.

If the number of characters input reaches the number passed in R1, further characters are ignored and cause Ctrl-G (ASCII 7) to be sent to OS_WriteC, which will normally cause a sound to be emitted. The deleting keys mentioned above will still function.

Alternatively – by setting the flags in R0 appropriately – you can reflect the character held in R4, instead of the characters typed. This is useful, for example, to read a password without echoing its actual characters to the screen.

You must not call OS_ReadLine from an interrupt or event routine.

Related SWIs

OS_WriteC (SWI &00), OS_ReadC (SWI &04), OS_Word 0 (SWI &07)

Related vectors

ReadLineV, WrchV

OS_ReadEscapeState (SWI &2C)

Check whether an escape condition has occurred

On entry

—

On exit

the C flag is set if an escape condition has occurred

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadEscapeState sets or clears the carry flag depending on whether escape is set or not. Once an escape condition has been detected (either through this call or, for example, with OS_ReadC), it should be acknowledged using OS_Byte 126 or cleared using OS_Byte 124.

This call is useful if a program is executing in a loop which the user may want to escape from, but isn't performing any input operations which would let it know about the escape.

Note that OS_ReadEscapeState may be called from an interrupt routine. However, OS_Byte 126 may not be, so if an escape is detected under interrupts, the interrupt routine must set a flag which is checked by the foreground task, rather than attempt to acknowledge the escape itself.

Related SWIs

OS_Byte 124 (SWI &06), OS_Byte 126 (SWI &06)

Related vectors

None

OS_InstallKeyHandler (SWI &3E)

Install a key handler or read the address of the current one

On entry

R0 =	0 to read address of current keyboard handler
	1 to read keyboard ID from keyboard (1 for UK keyboards)
	>1 to set address of new keyboard handler

On exit

R0 = address of current/old keyboard handler, or keyboard ID

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_InstallKeyHandler installs a new keyboard handler to replace the default code.

Related SWIs

None

Related vectors

None

*Commands

*Configure Caps

Sets the configured value for Caps Lock to ON

Syntax

```
*Configure Caps
```

Parameters

None

Use

*Configure Caps sets the configured value for Caps Lock to ON, so that when you switch on or reset your machine, you will start typing in capital letters. This is the default setting.

Example

```
*Configure Caps
```

Related commands

*Configure NoCaps, *Configure ShCaps

Related SWIs

OS_Byte 202 (SWI &06)

Related vectors

None

*Configure Delay

Sets the configured delay before keys start to auto-repeat

Syntax

```
*Configure Delay n
```

Parameters

n	delay (in centiseconds)

Use

*Configure Delay sets the configured delay before keys start to auto-repeat. A value of zero disables auto-repeat. The default value is 32.

Example

```
*Configure Delay 20
```

Related commands

*Configure Repeat

Related SWIs

OS_Byte 11 (SWI &06)

Related vectors

None

*Configure NoCaps

Sets the configured value for Caps Lock to OFF

Syntax

```
*Configure NoCaps
```

Parameters

None

Use

*Configure NoCaps sets the configured value for Caps Lock to OFF, so that when you switch on or reset your machine, you will start typing in lower case. Caps is the default setting.

Example

```
*Configure NoCaps
```

Related commands

*Configure Caps, *Configure ShCaps

Related SWIs

OS_Byte 202 (SWI &06)

Related vectors

None

*Configure Repeat

Sets the configured interval between the generation of auto-repeat keys

Syntax

```
*Configure Repeat n
```

Parameters

n	interval (in centiseconds)

Use

*Configure Repeat sets the configured interval between the generation of auto-repeat keys. A value of zero sets an infinite interval, so the character repeats just once, after the auto-repeat delay. To completely disable auto-repeat, set the Delay to zero.

The default value is 8.

Example

```
*Configure Repeat 3
```

Related commands

*Configure Delay

Related SWIs

OS_Byte 12 (SWI &06)

Related vectors

None

*Configure ShCaps

Sets the configured value for Caps Lock to ON, Shift producing lower case letters

Syntax

```
*Configure ShCaps
```

Parameters

None

Use

*Configure ShCaps sets the configured value for Caps Lock to ON, so that when you switch on or reset your machine, you will start typing in capital letters. Holding down the Shift key will produce lower case letters, which does not happen when Caps is the configured value. Caps is the default value.

Example

```
*Configure ShCaps
```

Related commands

*Configure NoCaps, *Configure Caps

Related SWIs

OS_Byte 202 (SWI &06)

Related vectors

None

*Key

Assigns a string to a function key

Syntax

```
*Key keynumber [string]
```

Parameters

keynumber	a number from 0 to 15
string	any GSTrans-compatible string

Use

*Key assigns a string to a function key. It provides a very simple way of setting up function keys so that repetitive or error-prone strings (such as complex commands) can be initiated with a single keystroke. You can use any string up to 255 characters long.

The string is transformed by GSTrans before being stored. This means that you can, for example, represent Return using '|M' (as in the example below). See the section on GSTrans for details.

The string is stored in the system variable Key$*keynumber*, for example Key$1 for function key 1. This enables a key's definition to be read before it is used, and manipulated like any other variable. Also, because a key string can be set as a macro, its value may be made to change each time it is used.

In addition to F1 to F12, these keys can act as function keys by default:

- Print as F0
- Insert as F13

and these keys can be made to act as function keys by the command *FX4,2:

- Copy as F11
- left arrow as F12
- right arrow as F13
- down arrow as F14
- up arrow as F15

Function keys are generally unaffected by a soft break, but lost following a hard break.

Example

```
*Key 8 *Audio On|M *Speaker On|m *Volume 127|m
*SetMacro Key$1 ||The time is <Sys$Time>|m
```

Related commands

*Set, *SetMacro

Related SWIs

None

Related vectors

None

24 The CLI

Introduction

There are two ways in which you can interact with the OS and the various modules which provide extensions to it. The first way is to call one of the many SWI routines provided, such as OS_Byte, OS_ReadMonotonicTime, Wimp_Initialise etc. The SWI interface provides an efficient calling mechanism for use within programs in any language.

However, for users wishing to issue commands to the operating system, the SWI interface is not so convenient. As it is difficult to remember SWI names, reason codes, register contents on entry and exit, etc, the *command line interpreter* (CLI) interface is often used. Using this technique, you enter a textual command string, possibly followed by parameters, which is then passed by the application to the OS. The OS tries to decode the command and carry out the appropriate action. If the command is not recognised by the OS, the other modules in the system try to execute the command instead.

The CLI interface is a powerful one because the OS performs a certain amount of pre-processing on the line before it attempts to interpret it. For example, variable names may be substituted in the parameter part of the line, and command aliases may be used.

By convention, an application passes commands to the OS if they are prefixed by the * character. For example, from the BASIC '>' prompt, any OS command may be issued simply by making * the first non-space character on the line. The * is not part of the command; the OS, in fact, strips any leading *s and spaces from a command before it tries to decode it.

Some languages also provide built-in statements which can be used to perform an OS command. Again, BASIC provides the OSCLI statement, which evaluates a string expression and passes this to the OS command line interpreter. The 'C' language provides the system() function for the same purpose.

Overview and Technical Details

A program can call the CLI using the SWI OS_CLI. This simply passes a string from the program to the CLI to be interpreted. If you wish to allow the user to type a number of CLI commands, then you can pass 'GOS', described in this chapter, as the string to OS_CLI. See the chapter entitled *Program Environment* on page 1-277, for information on how to set up RISC OS to return to your program when the user types *Quit.

CLI effects

When a CLI command is received by the kernel, it performs a number of operations upon it. Note that in most cases, the case of commands is ignored. Only if you are creating something with a name is the case kept. The sections below go through each of these.

Leading characters

Certain leading characters will be treated in a special way:

'*'	all leading stars are discarded
' '	all leading spaces are discarded
'|'	this indicates that the line is a comment, and will be ignored
'/'	treat the rest of the command as if it had been prefixed with *Run
'%'	skip alias checking
'-'	override current filing system name: eg -adfs-
'.'	check for Alias$. and use *Cat if it doesn't exist

Apart from '%' and '-', the above commands should be self-explanatory. '%' is used to access a built-in command that currently has an alias overriding it. See the section below on aliases.

Context overriding

The currently selected filing system can be overridden in two different ways. The command can be prefixed with -name- and name:, where name is the name of a filing system or module. That is, you supply an absolute name of the filing system or module to send the command to. This gets around the problem of having to select the other filing system, perform the command and then re-enter the original filing system. For example, if you are on the net and want to look at a file on the current adfs device, the sequence of commands:

```
*adfs
*Info Fred
*net
```

can be replaced with either:

```
*-adfs-Info Fred
```

or even more succinctly:

```
*adfs:Info Fred
```

Here are some examples of overrides:

```
*-net-cat
*SpriteUtils:Slist
*-Module#SpriteUtils-SInfo
```

Note that if you are using -net- or net:, you cannot specify nodes on the net: eg -net#spqr-. This is because the command prefix only alters the filing system selected for the command. The part of an object specification after the '#' character is not part of the filing system name but is part of the object name. For example, if you wish to issue a command such as:

```
*net#oz:info fred
```

you can use instead:

```
*net:info #oz:fred
```

Redirection

Normally, input comes from the keyboard and output goes to the screen. Redirection allows this source and destination to be changed to any file or device. Output redirection can be viewed as having a *Spool file open for the duration of the command, and disabling all streams except for that one. Input redirection is like having a *Exec file open for the duration of the command.

Here are the possible commands:

{ > *filename* }	Output goes to filename
{ < *filename* }	Input read from filename
{ >> *filename* }	Output appended to filename

A redirection command can appear anywhere in a line. Note that there must only be one space between all the elements in a redirection command or it will not be recognised as one. After being decoded, it is stripped before the rest of the command is interpreted. You can put as many redirection commands as you like on a line, however only the last one in a given direction will be acted on.

Here are some examples of redirection:

```
*Cat { > mycat }
*Lex { > printer: }
*BASIC -quit { < answers } prog
*fred { < infile > outfile }
*Cat { > out1 }{ < infile }{ > out2 }
```

The fourth example shows how redirections can be concatenated within the same pair of braces.

In the final example, `out1` will be created with nothing in it, input will be read from `infile` and output will go to `out2`.

Aliases

An alias is a variable of the form AliasScmd, where cmd is the command name to match. If an alias exists which matches the current * Command, the following takes place: the OS obtains the value of the variable and replaces any of %0 to %9 in the value by the parameters, separated by spaces, that it reads on the rest of the input line. %*n in an alias stands for the rest of the command line, from parameter 'n' onwards.

Any unused parameters, which are given, are directly appended to the alias. The OS then recursively calls OS_CLI for all lines in the expanded value. However, it may give up at this stage if either the stack or its buffer space becomes full. For example, suppose the command

```
*SetPS 0.235
```

is issued. Suppose further that a variable exists called AliasSSetPS, and that this has the value –NET–PS %0IMConfigure PS %0. The OS will match the command name against the alias variable. It will then substitute all occurrences of %0 in the variable's value by 0.235. Then, the two lines of the variable will be executed thus:

```
-NET-PS 0.235
Configure PS 0.235
```

So, the net effect of executing the original command is to set the network printer server both temporarily, and also in the permanent configuration.

Another example using the parameter substitution is

```
*Set Alias$Mode Echo |<22> |<%0>
```

The 'I's before the angle brackets are to stop them from being evaluated when the *Set command is entered. Typing `*Mode n` will then set the display to mode 'n'.

Look-up the command

After all the previous steps have been completed, the command that is left after pre-processing must be executed. This is a list in order of the things that RISC OS will check to execute a command:

- is it a command internal to RISC OS
- kernel checks the first module in turn to see whether it contains the command
- kernel moves onto the next module and so on until the end of the module list
- one of the modules is the filing system manager, File Switch, which has its command table checked by the kernel. The commands contained by this module are the commands that apply to all filing systems, such as *Cat.
- after the module search is complete, the kernel inspects the filing system specific commands in the current filing system module
- If the command is not recognised by the filing system module, the kernel issues an 'unknown command' service call. If the net is the current filing system, the command is sent to the file server, to see if the command is implemented on the fileserver. For example, *pass.
- if the command is still not recognised, then an attempt will be made to *Run it using the current path. The result of this *Run is passed back to the user.

Reading CLI parameters

If you are writing a module, the chances are that you will want to recognise one or more * Commands. The chapter entitled *Modules* on page 1-191 explains how you can cause the OS to recognise commands for you, and pass control to your module when one has been found. This section describes the OS calls which are available to facilitate the decoding of the rest of the command line.

The calls mentioned here may also be used by * Commands activated in other ways, eg a transient command loaded from disc. However, the way in which the tail of the command line is discovered will vary for these types of commands. See the chapter entitled *Program Environment* on page 1-277 for details.

On entry to your * Command routine, R0 contains a pointer to the 'tail' of the command, ie the first character after the command name itself (with spaces skipped). R1 contains the number of parameters, where a parameter is regarded as a sequence of characters separated by spaces.

The way in which the command uses the parameters depends on what it is doing. First, if there are too many or too few parameters, an error could be given. (A module can arrange for the OS to do this automatically.)

If a parameter is to be regarded as a string, OS_GSTrans may be used to decode any special sequences, eg control codes, variable names etc. If the parameter is a number, OS_ReadUnsigned might be used to convert it into binary. Finally, OS_EvaluateExpression could be used to read a whole arithmetic or string expression, and return the result in a buffer.

These calls are documented in the chapter entitled *Conversions*, along with other useful conversion routines such as OS_ReadUnsigned.

Note that the convention on the Archimedes is to have parameters separated by spaces. Some of the built-in commands which have been carried over from the BBC/Master machines also allow commas. You should not support this option.

SWI Calls

OS_CLI (&05)

Process a supervisor command

On entry

R0 = pointer to string terminated by Null, Linefeed or Return

On exit

R0 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not-re-entrant

Use

OS_CLI will execute a string passed to it as if it had been typed in at the supervisor command line. When it is called, it performs the following actions:

Check stack space

The OS needs a certain amount of workspace to deal correctly with a command. If this is not available, the error `No room on supervisor stack` will be generated.

Check command length

A * Command line must be less than or equal to 256 bytes long, including the terminating character. If it is not, the line is ignored. No error is generated.

Execute command

The command is then executed as any other * Command. This is described in the technical description.

Related SWIs

None

Related vectors

CLIV

OS_ChangeRedirection (SWI &5E)

Read or write OS_CLI input/output redirection handles

On entry

R0 = new file handle for input
  0 = not redirected
  –1 = leave alone
R1 = new file handle for output
  0 = not redirected
  –1 = leave alone

On exit

R0 = old file handle for input
  0 = not redirected
R1 = old file handle for output
  0 = not redirected

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI reads or writes the file handles used by OS_CLI to redirect input/output. It is mainly provided for the use of the Task Manager, but you may also find the call useful.

Related SWIs

None

Related vectors

None

* Commands

*GOS

Calls Command Line mode, and hence allows you to type * Commands

Syntax

*GOS

Parameters

None

Use

*GOS starts the RISC OS Supervisor application from the current environment. The supervisor can only execute *Commands.

This is useful for entering simple commands for immediate execution, or for testing longer sequences of commands – while building command line scripts –on a line-by-line basis.

However you should be careful when calling it from the middle of an application which does not 'shell' new applications. For example, calling *GOS in the middle of writing a BASIC program will mean that you will lose all of your unsaved work.

From the desktop, pressing F12 has a similar effect. To return to the desktop, press Return at the start of a line with the Supervisor prompt ('*'). If you do not have this prompt, you will first have to type *Quit to leave the application you are using.

See the section entitled *Overview and Technical Details* on page 2-430 for a description of how the command line interface works.

Related commands

*Quit, *Desktop

Related SWIs

None

Related vectors

None

25 The rest of the kernel

Introduction

Kernel commands are covered here that do not merit a chapter by themselves.

The following SWIs are described:

• OS_Byte 0	display OS version information
• OS_Byte 1	write user flag
• OS_Byte 241	read/write user flag
• OS_HeapSort (SWI &4F)	a fast and memory efficient sorting routine
• OS_Confirm (SWI &59)	get a yes or no answer to a question
• OS_CRC (SWI &5B)	calculate a cyclic-redundancy check for a block
• IIC_Control (SWI &240)	control of external IIC devices

The following * Commands are also described:

• *Configure Language	select the language to use at power on
• *Help	get help on commands

SWI Calls

OS_Byte 0 (SWI &06)

Display OS version information

On entry

R0 = 0
R1 = 0 to display message, or other value to return result

On exit

R0 preserved
R1 = OS version number if R1 non-zero on entry
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

If this is called with R1=0, an error is produced, and the text of the error shows the version number and creation date of the operating system. If it is called with R1 ≠ 0, then a version-dependent result is returned in R1.

Related SWIs

None

Related vectors

ByteV

OS_Byte 1 (SWI &06)

Write user flag

On entry

R0 = 1
R1 = new value

On exit

R0 preserved
R1 = previous value
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 OS_Byte accesses a location which is guaranteed to be unused by the OS. You can use this to pass results between programs. However, system variables provide much more versatile means of doing this. The byte may also be read and written using OS_Byte 241.

Related SWIs

OS_Byte 241 (SWI &06)

Related vectors

ByteV

OS_Byte 241 (SWI &06)

Read/write user flag

On entry

R0 = 241
R1 = 0 to read, or new value to write
R2 = 255 to read, or 0 to write

On exit

R0 preserved
R1 = value before being overwritten
R2 corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The value stored is changed by being masked with R2 and then exclusive ORd with R1: ie ((value AND R2) EOR R1). This means that R2 controls which bits are changed and R1 supplies the new bits.

This OS_Byte accesses a location which is guaranteed to be unused by the OS. You can use this to pass results between programs. However, system variables provide much more versatile means of doing this. The byte may also be written to using OS_Byte 1.

Related SWIs

OS_Byte 1 (SWI &06)

Related vectors

ByteV

OS_HeapSort (SWI &4F)

Heap sort a list of objects

On entry

R0 = number of elements to sort
R1 = pointer to array of word size objects, and flags in top 3 bits
R2 = type of object (0 - 5), or address of comparison routine
R3 = workspace pointer for comparison procedure (only needed if R2 > 5)
R4 = pointer to array of objects to be sorted (only needed if flag(s) set in R1)
R5 = size of an object in R4 (only needed if flag(s) set in R1)
R6 = address of temporary workspace of R5 bytes
(only needed if R5 > 16k or bit 29 of R1 is set)

On exit

R0 - R6 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI will sort a list of any objects using the heap sort algorithm. Details of this algorithm can be found in:

Sorting and Searching D.E. Knuth (1973) Addison-Wesley, Reading Massachusetts, pages 145 -149.

It is not as fast as a quicksort for average sorts, but uses no extra memory than that which is initially passed in.

Basic usage

Used in the simplest way, only R0, R1 and R2 need be set up. R0 contains the number of objects that are in the list. R1 points to an array of word-sized entries. The value of R2 controls the interpretation of this array:

R2 value	Treat R1 as pointing to an array of...
0	cardinal (unsigned integer)
1	integer
2	pointer to cardinal
3	pointer to integer
4	pointer to characters (case insensitive)
5	pointer to characters (case sensitive)
>5	pointer to custom object In this last case, R2 is the address of the comparison routine

Comparison routine

If the R2 value is less than 6, then this call will handle sorting for you. If you want to sort any other kind of object, then you must provide a routine to compare two items and say which is the greater. Using this technique, any complex array of structures may be sorted. If you wish to use a comparison routine, then R2 contains the address of it. R3 must be set up with a value, usually a workspace pointer.

When called, the comparison routine is entered in SVC mode, with interrupts enabled. R0 and R1 contain two objects from the array passed to this SWI in R1. What they represent depends on what the object is, but in most cases they would be pointers to a structure of some kind. R12 contains the value originally passed in R3 to this SWI. Usually this is a workspace pointer, but it is up to you what it is used for.

Whilst in this routine, R0 - R3 may be corrupted, but all other registers must be preserved. The comparison routine returns a less than state in the flags if the object in R0 is less than the object in R1. A greater or equal state must be returned in the flags if the object in R0 is greater than or equal to the object in R1.

Advanced features

In cases where R2 is greater than 1, then there are two arrays in use. The word sized array of pointers pointed to by R1 and the 'real' object array. You can supply the address of this real array in R4 and the size of each object in it in R5. If this is done, then a number of optional actions can be performed. The top bits in R1 can be used as follows:

Bit	Meaning
29	use R6 as workspace
30	build word-array of pointers pointed to by R1 from R4,R5
31	sort true objects pointed to by R4 after sorting the pointers

Bit 30 is used to build the pointer array pointed to by R1 using R4 and R5 before sorting is started. It will create an array of pointers, where the first pointer points to the first object, the second pointer to the second object and so on. After sorting, these pointers will be jumbled so that the first pointer points to the 'lowest' object and so on.

Bit 31 is used to sort the real objects pointed to by R4 into the order described by the pointers in the array pointed to by R1 after sorting is complete. It may optionally be used in conjunction with bit 30.

If the size in R5 is greater than 16 Kbytes or if bit 29 is set in R1, then a pointer to workspace must be passed in R6. This points to a block R5 bytes in length. One reason for setting bit 29 is that this SWI will otherwise corrupt the RISC OS scratch space.

Related SWIs

None

Related vectors

None

OS_Confirm (SWI &59)

Get a yes or no answer

On entry

—

On exit

R0 = key that was pressed, in lowercase
the C flag is set if an escape condition occurred
the Z flag is set if the answer was Yes

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI gets a yes or no answer from the user. If the mouse pointer is visible, then it changes it to a three button mouse shape. The left button indicates yes, while the other two indicate no. On the keyboard, a key appropriate to the territory indicates yes, and any other key indicates no.

You should always check whether the answer was yes or no by testing the Z flag, rather than the value returned in R0; this ensures that your program will not need modifying for use with different territories.

The result in R0 is returned in lowercase, irrespective of the keyboard state. It is made available should you need to reflect a character to the screen.

An escape condition will abort the SWI and return with the C flag set.

Related SWIs

None

Related vectors

None

OS_CRC (SWI &5B)

Calculate the cyclic-redundancy check for a block of data

On entry

R0 = CRC continuation value, or zero to start
R1 = pointer to start of block
R2 = pointer to end of block
R3 = increment (in bytes)

On exit

R0 = CRC calculated
R1 - R3 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI calculates the cyclic-redundancy check value for a block of data. This is used to check for errors when, for example, a block of data is stored on a disk (although ADFS doesn't use this call) or sent across a network and so on. If the CRC calculated when checking the block is different from the old one, then some errors are in the data.

The block described in R1 and R2 is exclusive. That is, the calculation adds R3 to R1 each step until R1 equals R2. If they never become equal, then it will continue until crashing the machine. For example R1=100, R2=200, R3=3 will never match R1 with R2 and is not permitted.

The value of the increment in R3 is the unit that you wish to use for each step of the CRC calculation. Usually, it would be 1, 2 or 4 bytes, but any value is permitted. Note that the increment can be negative if you require it.

Related SWIs

None

Related vectors

None

IIC_Control (SWI &240)

Control IIC devices

On entry

R0 = device address (bit 0 = 0 to write, bit 0 = 1 to read)
R1 = pointer to block
R2 = length of block in bytes

On exit

R0 - R2 preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call allows reading and writing to IIC devices. IIC is an internal serial protocol. It is used in RISC OS machines for writing to the clock chip and IIC compatible devices on expansion cards.

The possible error is 'No acknowledge from IIC device' (&20300).

Related SWIs

None

Related vectors

None

*Commands

*Configure Language

Sets the configured language used at power on

Syntax

```
*Configure Language module_no
```

Parameters

module_no	the module number of the language which will be started after power on. The default language is the desktop.

Use

*Configure Language sets the configured language used at power on by specifying its module number. Use the *Modules command to check the number of the required language, especially if you have added or removed modules. You should also be aware that module numbers may differ between versions of RISC OS.

Note that the configured language is not entered if a boot file is run at power on using *Exec. Instead, you must select the language at the end of the boot file, or use an Obey file.

Example

```
*Configure Language 0
```
*Starts up in Command Line mode, with * prompt*

Related commands

*Modules

Related SWIs

None

Related vectors

None

*Help

Gives brief information about each command

Syntax

```
*Help [keyword]
```

Parameters

keyword	the command name(s) to get help on

Use

*Help gives brief information about each command in the machine operating system, including its syntax. It also has help on some special keywords:

`*Help Commands`	lists all the available utility commands
`*Help FileCommands`	lists all the commands relating to filing systems
`*Help Modules`	lists the names of all currently loaded modules, with their version numbers and creation dates
`*Help Station`	displays the current network and station numbers of your machine
`*Help Syntax`	explains the format used for syntax messages

The usual use of *Help is to confirm that a command is appropriate for the job required, and to check on its syntax (the number, type and ordering of parameters that the command requires). When you issue the *Help command at the normal Command Line prompt, 'paged mode' is switched on: the computer displays a screenful of text, then waits until you press Shift before moving on.

Example

The specification of the keyword can include abbreviations to allow groups of commands to be specified. For example,

`*Help Con.`	*produces information on *Configure and *Continue*
`*Help .`	*gives help on all subjects*

Related commands

None

Related SWIs

None

Related vectors

None