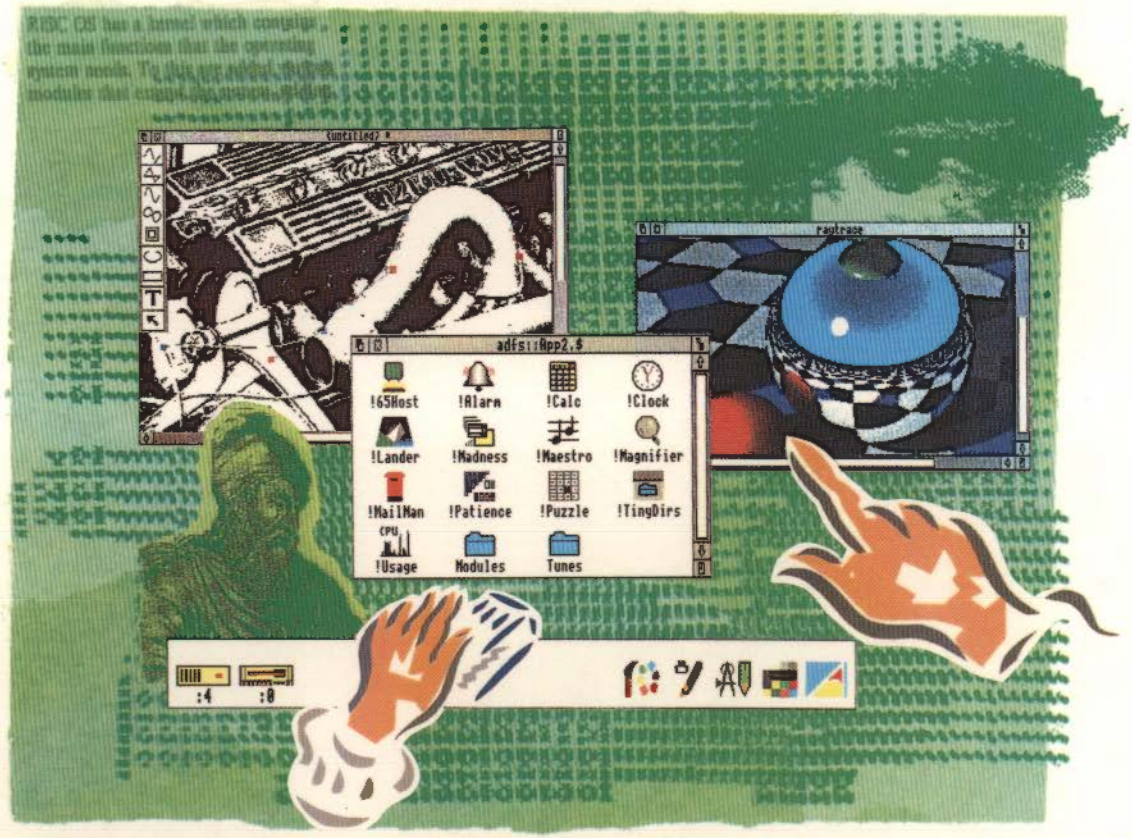
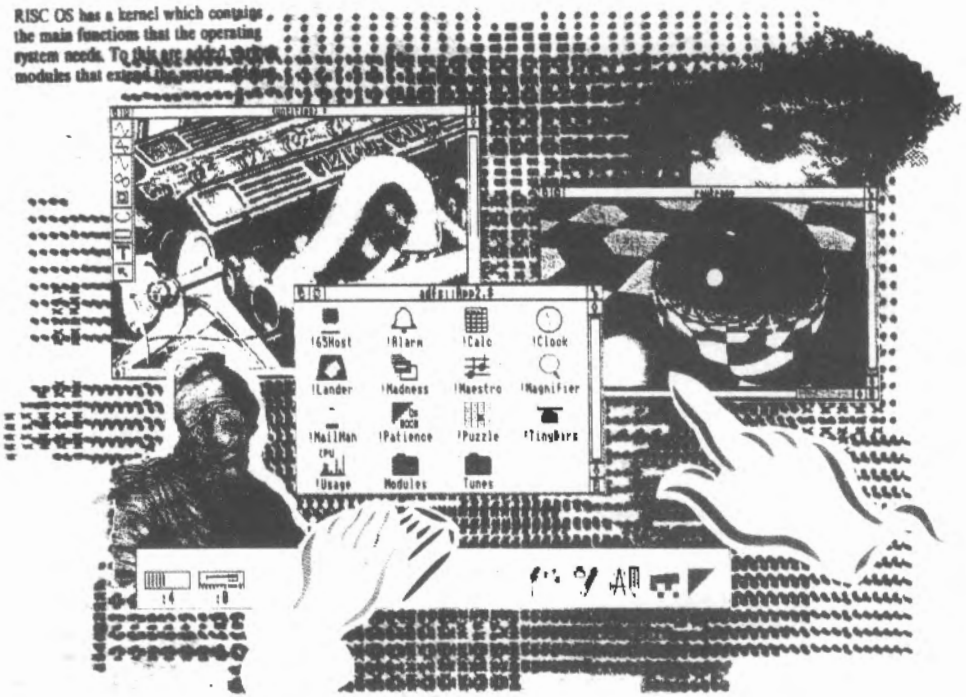
RISC OS
PROGRAMMER'S REFERENCE MANUAL
Volume II



Acorn
The choice of experience.

RISC OS
PROGRAMMER'S REFERENCE MANUAL
Volume II



Acorn
The choice of experience.

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

All correspondence should be addressed to:

Customer Service
Acorn Computers Limited
Fulbourn Road
Cambridge CB1 4JN

Information can also be obtained from the Acorn Support Information Database (SID). This is a direct dial viewdata system available to registered SID users. Initially, access SID on Cambridge (0223) 243642: this will allow you to inspect the system and use a response frame for registration.

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

Edition 1
Published 1989: Issue 1
ISBN 1 85250 061 1
Published by Acorn Computers Limited
Part number 0483,021

Contents

In this volume

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*. 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 *Technical Details* section 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 programs 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 *Program Environment* chapter. 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. (Refer to the description of events in the introductory part of this manual.)

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.

***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* in the introductory part 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 the Table entitled *Character sets*.

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 or 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 enabl 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 Key$ system variable which corresponds to the function key. The function key number is the lower nibble of the character. Thus, if the character is &81, the variable read is Key$1.

The variable refers to a string, which is copied into the function key buffer. If the string was a null string (the function key wasn't set), then RISC OS continues, removing the next character from the input buffer.

Otherwise, the first character is removed from the function key buffer and returned to the calling program. Characters read from this buffer are returned without interpretation in any way.

Subsequent calls to OS_ReadC and OS_Byte 129 spot that a function key is being read, and remove characters from the function key buffer instead of looking in the input buffer. This continues until the last character has been read from the buffer. Input then reverts to the normal input buffer.

OS_Byte 216 is used to see how much of a function key string remains to be read from the function key buffer. It can also change this value, to terminate for instance, but must be used with care.

Setting and clearing

To set a function key, a number of commands can be called:

- *Key <n> <string>
- *Set Key$n <string>

- *SetMacro Key$n <expression>This is passed through OS_GSTrans when it is copied to the function key buffer. This is interesting because it means that the string generated by a function key can change every time it is used.

To reset one or more function keys, there is also a variety of commands that can be used:

- *Key <n> will reset function key n
- *Unset Key$n same effect
- *Unset Key$*will reset all function keys
- OS_Byte 18 same effect

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.

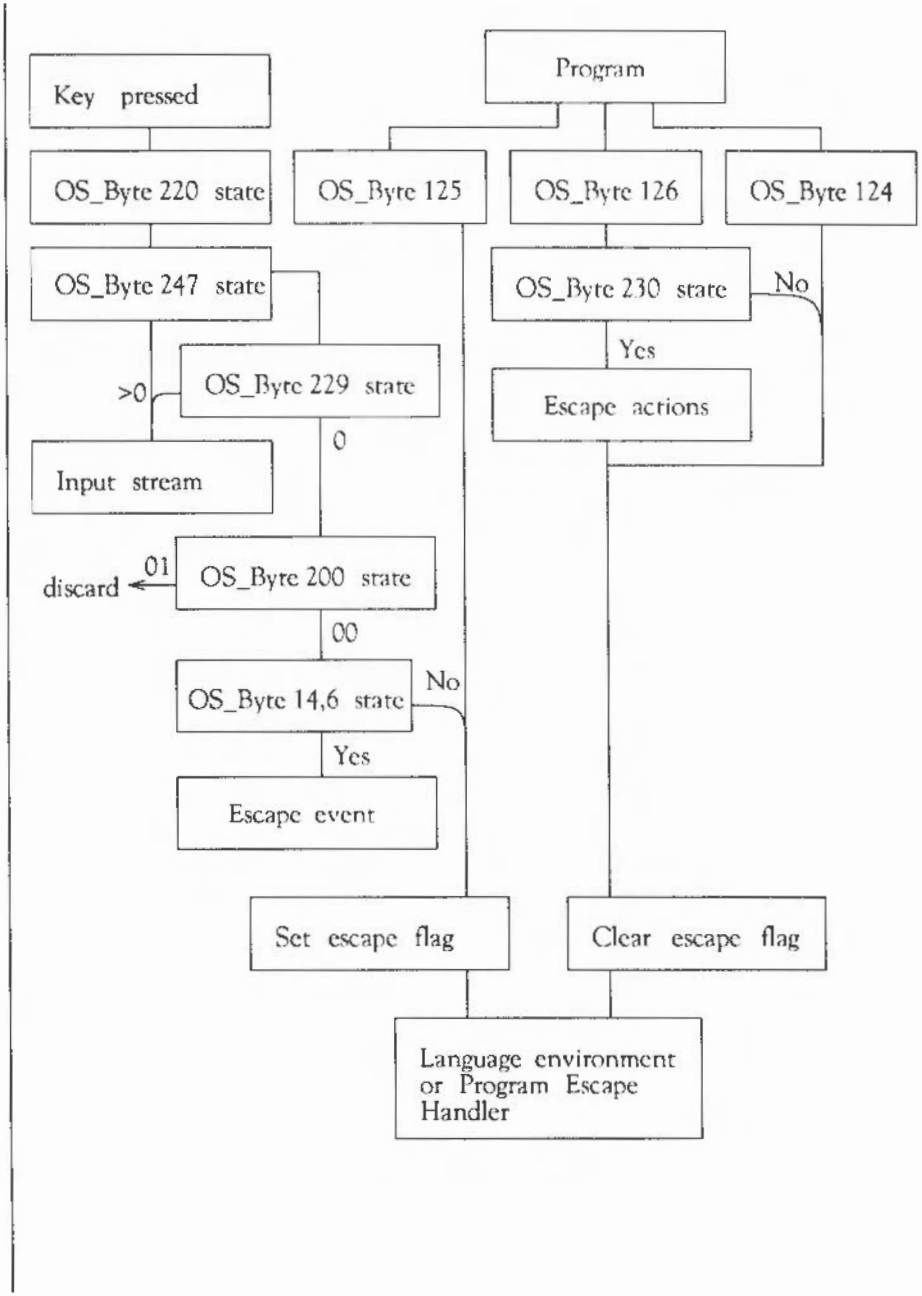
OS_Byte 253 can be used to see what kind of reset the last one was.

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

On the next page is a diagram illustrating how all the calls in the escape system work together. A description of this interaction follows the diagram.


Program
Key pressed
OS_Byte 220 state
OS_Byte 125
OS_Byte 126
OS_Byte 124
OS_Byte 247 state
OS_Byte 230 state
No
Yes
>0
OS_Byte 229 state
Escape actions
0
Input stream
discard
01
OS_Byte 200 state
00
OS_Byte 14,6 state
No
Yes
Escape event
Set escape flag
Clear escape flag
Language environment or Program Escape Handler

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 Port

Input buffer

The serial driver will attempt to stop the sender transmitting when the amount of free space in the serial input buffer falls below a set level. The idea is that this space gives enough time for the sender to recognise the command and stop without overflowing the buffer. OS_Byte 203 can change the setting of this level.

Baud rates

The baud rate of the serial input can be changed with OS_SerialOp 5 or OS_Byte 7. It can be read with OS_SerialOp5, or with OS_Byte 242, which is described in the chapter entitled *Character output*.

Control

The description of the serial port in the chapter entitled *character output* covers the handshaking control of the serial port.

OS_Byte 204 can be used to stop any incoming data being buffered by the serial driver. The port is still active, and serial errors can still occur, but the data is discarded.

OS_Byte 181 allows the data that comes in from a serial port to be acted on by RISC OS as if it had been typed at the keyboard.

Direct reading

OS_SerialOp 4 allows directly getting a byte from the serial input buffer. This means that input from the keyboard could be read from the main input stream concurrently. Thus, a means of having two separate channels of input would exist.

*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

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 `*FX 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

ReadCV

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.

The write command can also be performed by `*FX 2,<value>`

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:

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:

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.

This command can also be performed by `*FX 4,<state>`

Related SWIs

OS_Byte 237 (SWI &06)

Related vectors

ByteV

OS_Byte 7 (SWI &06)

Write serial port receive rate

On entry

R0 = 7
R1 = baud rate code

On exit

R0 = preserved
R1 = corrupted
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 serial port baud rate for receiving data as follows:

Value	Baud rate
0	9600
1	75
2	150
3	300
4	1200
5	2400
6	4800
7	9600
8	19200
9	50
10	110
11	134.5
12	600
13	1800
14	3600
15	7200

The settings from 0 to 8 are in an order compatible with earlier operating systems. The other speeds from 9 to 15 provide all the other standard baud rates.

The default rate is that set by `*Configure Baud`.

This command can also be performed by `*FX 7,<baud rate>`

Related SWIs

OS_Byte 8 (SWI &06)

Related vectors

ByteV

OS_Byte 11 (SWI &06)

Write keyboard auto-repeat delay

On entry

R0 = 11 (SWI &0B)
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.

The write command can also be performed by `*FX 11,<delay>`

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 (&0C)
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.

The operation can also be performed by `*FX 12,<rate>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 18 (SWI &06)

Reset function key definitions

On entry

R0 = 18 (&12)

On exit

R0 = preserved
R1 = corrupted
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.

This operation can also be performed by `*FX 18`

Related SWIs

None

Related vectors

ByteV

OS_Byte 118 (SWI &06)

Reflect keyboard status in LEDs

On entry

R0 = 118 (&76)

On exit

R0 = preserved
R1 = corrupted
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 in this chapter 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.

This operation can also be performed by `*FX 118`

Related SWIs

None

Related vectors

ByteV

OS_Byte 120 (SWI &06)

Temporarily lock auto-repeat

On entry

R0 = 120 (&78)
R1 = 0
R2 = 0

On exit

R0 = preserved
R1 = corrupted
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.

This command can also be performed by `*FX 120`

Related SWIs

None

Related vectors

ByteV

OS_Byte 121 (SWI &06)

Keyboard scan

On entry

R0 = 121 (&79)
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 tables in the *Technical Details* of this chapter 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 (&7A)

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 tables in the *Technical Details* of this chapter for a complete list). All key numbers below 16 are ignored. This excludes all Shift, Ctrl, Alt and mouse keys. It is equivalent to 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 (&7C)

On exit

R0 = preserved
R1 = corrupted
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 and returns, without calling the escape handler.

This command can also be performed by `*FX 124`

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 (&7D)

On exit

R0 = preserved
R1 = corrupted
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.

This command can also be performed by `*FX 125`

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 (&7E)

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.

This command can also be performed by `*FX 126`

Related SWIs

OS_Byte 124 (SWI &06), OS_Byte 125 (SWI &06)

Related vectors

ByteV

OS_Byte 129 (SWI &06)

Read keyboard for information

On entry

R0 = 129 (&81)
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) or &A1 (RISC OS 2.0)
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.

The relationship of keys to internal key numbers is the keyboard scan numbers in BBC/Master series computers but is different for other Acorn series computers.

A list of all internal key numbers can be found in the *Technical Details* section of this chapter.

Related SWIs

None

Related vectors

ByteV

OS_Byte 177 (SWI &06)

Read input stream selection

On entry

R0 = 177 (&B1)
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 that character input gets its characters from:

•

It must never be altered with this call by changing the 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 (&B2)
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 181 (SWI &06)

Read/write serial input interpretation status

On entry

R0 = 181 (&B5)
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 = NoIgnore state (see OS_Byte 182 in the *Character output* chapter)

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.

Usually, top-bit-set characters read from the serial input buffer are not treated specially. For example, if the remote device sends the code &85, when this is read, using OS_ReadC for example, that ASCII code will be returned to the caller immediately. It is sometimes useful to be able to treat serial input characters in exactly the same way as keyboard characters. OS_Byte 181 allows this.

The state value passed to this call has two values:

0 In this state the keyboard interpretation is placed on characters read from the serial input buffer.

1 This is the default state in which no keyboard interpretation is done. This means that:
- the current escape character is ignored,
- the function key codes are not expanded,
- events are not generated.

The write command can also be performed by `*FX 181,<state>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 196 (SWI &06)

Read/write keyboard auto-repeat delay

On entry

R0 = 196 (&C4)
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.

The write command can also be performed by `*FX 196,<delay>`

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 (&C5)
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).

The write command can also be performed by `*FX 197,<rate>`

Related SWIs

OS_Byte 11, OS_Byte 12 (SWI &06)

Related vectors

ByteV

OS_Byte 198 (SWI &06)

Read/write *Exec file handle

On entry

R0 = 198 (&C6)
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.

The write command can also be performed by `*FX 198,<handle>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 200 (SWI &06)

Read/write Break and Escape effect

On entry

R0 = 200 (&C8)
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.

The write command can also be performed by `*FX 200,<state>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 201 (SWI &06)

Read/write keyboard disable flag

On entry

R0 = 201 (&C9)
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.

The write command can also be performed by `*FX 201,<flag>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 202 (SWI &06)

Read/write keyboard status byte

On entry

R0 = 202 (&CA)
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	0	off	Alt
	1	on	
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.

The write command can also be performed by `*FX 202,<status>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 203 (SWI &06)

Read/write serial input buffer minimum space

On entry

R0 = 203 (&CB)
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 = serial ignore flag (see OS_Byte 204)

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.

The serial input routine attempts to halt input when the amount of free space left in the input buffer falls below a certain level. This call allows the value at which input is halted to be read or changed.

OS_SerialOp 0 can be used to examine or change the handshaking method.

The default value is 9 characters.

The write command can also be performed by `*FX 203,<value>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 204 (SWI &06)

Read/write serial ignore flag

On entry

R0 = 204 (&CC)
R1 = 0 to read or new flag 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 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 is used to read or change the flag which indicates whether serial input is to be buffered or not. Although this call can stop data being placed in the serial input buffer, data is still received by the serial driver. Errors will still generate events unless they have been disabled by OS_Byte 13.

If the flag is zero, then serial input buffering is enabled. Any non-zero value disables it.

The write command can also be performed by `*FX 204,<flag>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 216 (SWI &06)

Read/write length of function key string

On entry

R0 = 216 (&D8)
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.

The write command can also be performed by `*FX 216,<length>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 219 (SWI &06)

Read/write Tab key value

On entry

R0 = 219 (&DB)
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.

The write command can also be performed by `*FX 219,<value>`

Related SWIs	None
Related vectors	ByteV

OS_Byte 220 (SWI &06)

Read/write escape character

On entry

R0 = 220 (&DC)
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

The write command can also be performed by `*FX 220,<value>`

Related SWIs

None

Related vectors

ByteV

OS_Bytes 221 - 228 (SWI &06)

Read/write interpretation of buffer codes

On entry

R0 = 221 - 228 (&DD - &E4)
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

The write command can also be performed by `*FX <221 - 228>,<value>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 229 (SWI &06)

Read/write Escape key status

On entry

R0 = 229 (&E5)
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.

The write command can also be performed by `*FX 229,<status>`

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 (&E6)
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.

The write command can also be performed by `*FX 230,<status>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 237 (SWI &06)

Read/write cursor key status

On entry

R0 = 237 (&ED)
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.

The write command can also be performed by `*FX 237,<state>`

Related SWIs

OS_Byte 4 (SWI &06)

Related vectors

ByteV

OS_Byte 238 (SWI &06)

Read/write numeric keypad interpretation

On entry

R0 = 238 (&EE)
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.

The write command can also be performed by `*FX 238,<status>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 247 (SWI &06)

Read/write Break key actions

On entry

R0 = 247 (&F7)
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
0	Act as Reset
01	Act as escape key
10	No effect
11	Undefined

The default is 2_00000001, so Break causes an escape condition.

The write command can also be performed by `*FX 247,<value>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 253 (SWI &06)

Read last reset type

On entry

R0 = 253 (&FD)
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 (&FE)
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

The write command can also be performed by `*FX 254,<value>`

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
R1 = size of buffer
R2 = lowest ASCII value to pass
R3 = highest ASCII value to pass

On exit

R0 = corrupted
R1 = length of buffer read, not including Return.
R2 = corrupted
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 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 charactrs 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.

OS_ReadLine must not be called from an interrupt or event routine.

Related SWIs

OS_ReadC (SWI &04), OS_Word 0 (SWI &07), OS_WriteC (SWI &00)

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

OS_SerialOp 4 (SWI &57)

Get a byte from the serial buffer

On entry

R0 = 4

On exit

R0 is preserved
if C flag = 0 then R1 = character received
if C flag = 1 then R1 preserved.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call removes a character from the serial input buffer if one is present. If removing a character leaves the input buffer with more free spaces than are specified by OS_Byte 203, then transmission is re-enabled in the way specified by the state set by reason code 0.

Note that reception must have been enabled using OS_Byte 2 before this call will have any effect.

For a general description of OS_SerialOp, see the chapter entitled *character output*.

Related SWIs

None

Related vectors

None

OS_SerialOp 5 (SWI &57)

Read/write RX baud rate

On entry

R0 = 5
R1 = –1 to read or 0 - 15 to set to a value

On exit

R0 is preserved
R1 = old receive baud rate

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 has the same effect as an OS_Byte 7.

The value that is passed in R1 uses the same table of baud rates as this OS_Byte.

For a general description of OS_SerialOp, see the chapter entitled *character output*.

Related SWIs

OS_Byte 7 (SWI &06)

Related vectors

None

*Configure Caps

Configures Caps Lock ON

Syntax `*Configure Caps`

Parameters None

Use This command configures Caps Lock to be on, so that when you switch on or reset, 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

Configures the delay before keys start to auto-repeat

Syntax

```
*Configure Delay <n>
```

Parameters

<n> 0 to 255

Use

This command specifies the configured keyboard auto-repeat delay in centiseconds. 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

Configures Caps Lock OFF

Syntax

```
*Configure NoCaps
```

Parameters

None

Use

This command configures Caps Lock to be off, so that when you switch on or reset, 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> 0 to 255

Use

This command specifies the configured keyboard auto-repeat interval in centiseconds. 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.

This option can also be set from the desktop, using the Configure application.

Example

```
*Configure Repeat 3
```

Related commands

*Configure Delay

Related SWIs

OS_Byte 12 (SWI &06)

Related vectors

None

*Configure ShCaps

Configures Caps Lock ON, Shift producing lower case letters

Syntax

```
*Configure ShCaps
```

Parameters

None

Use

This command configures Caps Lock to be on, so that when you switch on or reset, 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

*Exec

Executes a command file

Syntax

```
*Exec [<pathname>]
```

Parameters

`<pathname>` a valid pathname specifying a file

Use

*Exec <pathname> opens the specified file for input. This command is mainly used for executing a list of operating system commands contained in a command file. The file, once open, takes priority over the keyboard or serial input streams.

*Exec with no parameter closes the exec file.

Example

```
*Exec !Boot
```

Related commands

*Obey

Related SWIs

OS_Byte 198 (SWI &06)

Related vectors

None

*Key

Assigns a string to a function key

Syntax

```
*Key <keynumber> [<value>]
```

Parameters

`<keynumber>`	number from 0 to 15
`<value>`	any GSTrans-compatible string

Use

This command assigns a string to a function key. Any string up to 255 characters long can be used.

The text is transformed by GSTrans before being stored. This means that you can represent Return using '|M' as in the example below. See the section on GSTrans for details.

The string is stored in a system variable, Key$<number>, 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

Time and Date

Introduction

There are two basic aspects of time dealt with in this chapter. Passive aspects, such as reading various clock settings and active ones, where an event occurs when a given time is reached. In this chapter, a clock is a place where a stored value is incremented on a regular basis. The time is the name of the value as it is read or written.

There are several clocks that increment every 1/100th of a second (centisecond). One of them cannot be changed except by a hard reset. This is useful for time-stamping events, such as mouse moves. Another can be changed by a program, so is useful for elapsed time calculations.

The real time clock keeps the real-world time, and represents time in centiseconds since 00:00:00 on January 1 1900. There are calls to present this information in a number of ways. The real-time can be converted to a string with complete program control over its format.

A variety of timer events can be set up. There are SWIs that will call your application after a given delay has passed or every time that delay has elapsed. You can set up a routine to sit on the ticker vector, to enable it to be called every centisecond.

A specialised form of timer event is one that will occur every time the screen driving hardware reaches the bottom of the screen. This event is useful for flicker-free redrawing. See the chapter entitled *VDU drivers* for further details.

Overview and Technical Details

There are four timers, which increment at a centisecond rate. They are:

- the monotonic timer (read-only)
- the system timer (read/write)
- the interval time (read/write)
- the real-time clock (read only, in general. ie. only users should change it)

Monotonic timer

A monotonic timer cannot be written, except by a hard reset or when the machine is turned on. OS_ReadMonotonicTime (SWI &42) allows you to read this value. It is useful for time-stamping within an application, such as event times. Because it can never be changed, the order of events cannot be confused.

It is stored as a 4-byte value with least significant byte first. It is incremented every centisecond, which means that it would take nearly 500 days for it to wrap around.

System clock

The system clock is stored as a 5-byte value. Like the monotonic timer it is reset by hard resets and increments every centisecond. However it can be altered. This is useful for measuring elapsed times in an application. OS_Word 1 reads the value and OS_Word 2 writes it.

Real-time

The real-time clock is stored as a 5-byte value in the CMOS clock chip and reflects the normal usage of the word clock. That is, it stores an elapsed time since 00:00:00 on January 1 1900. It can be set using the Clock or Alarm applications on the desktop.

A soft-copy of the real time clock is also kept by RISC OS and is used by the filing system to date-stamp files. This soft-copy is updated from the CMOS clock chip following a hard reset

Standard format

*Time will display the time and date from the CLI. It uses OS_Word 14,0 to display the information. Here is an example of the format that they present it in:

Tue,28 Mar 1989.13.25.54

5-byte format

The real-time can be read in the standard 5-byte format using OS_Word 14,3. This, or any, 5-byte time can be converted into the standard time string using OS_ConvertStandardDateAndTime (SWI &C0).

Changing real-time

The real-time time can be altered with OS_Word 15,8, the date with OS_Word 15,15, or both with OS_Word 15,24. These calls all use the time in a fixed string format the same as that above.

Format field names

The above standard time string is not flexible. To allow programs and users to customise the way that the time and date is presented, it is possible to supply a format string. The string is copied character for character to the output buffer unless a '%' is found. If this character is followed by any of the following codes, then the appropriate value copied to the output buffer.

Name	Value	Example
CS	Centi-seconds	99
SE	Seconds	59
MI	Minutes	05
12	Hours in 12 hour format	07
24	Hours in 24 hour format	23
AM	'am' or 'pm'	PM
PM	'am' or 'pm'	AM
WE	Weekday, in full	Thursday
W3	Weekday, in three characters	Thu
WN	Weekday, as a number	5
DY	Day of the month	01
ST	"st", "nd", "rd" or "th"	st
MO	Month name, in full	September
M3	Month name, in three characters	Sep
MN	Month as a number	09
CE	Century	19
YR	Year within century	87
WK	Week of the year, Mon to Sun	52

DN	Day of the year	364
0	Insert an ASCII 0 zero byte	
%	Insert a '%'	

To cause leading zeros to be omitted, prefix the field with the letter Z. For example, %zmn means the month number without leading zeros. %0 may be used to split the output into several zero-terminated strings.

For example, the standard time string would be produced using the following format string:

```
%W3,%DY %M3 %CE%YR.%24:%MI:%SE
```

OS_ConvertDateAndTime (SWI &C1) will convert a 5-byte time into a string using a supplied format string.

BCD conversions

The CMOS clock chip stores the time internally in a Binary Coded Decimal (BCD) format. OS_Word 14,1 will read the time as a 7-byte BCD block. OS_Word 14,2 will convert this BCD block into the standard string format.

Timer events

There are three different causes of timer events: the interval timer, the timer chain and the VSync timer.

Interval timer

The interval timer is a 5-byte clock that increments every centisecond. If enabled by OS_Byte 14, an event will occur when the counter reaches zero. Thus to wait for a given time, the interval timer must be set to the negative of it using OS_Word 4. OS_Word 3 can read the current setting of the interval timer.

For example, to wait 10 seconds, –1000 must be passed to OS_Word 4.

The interval timer is kept for compatibility with earlier Acorn operating systems. Its use should be avoided if possible. It is especially important that this is not used under the Wimp, since it is cannot cope with more than one program using it at once.

Timer chain

An easier to use and more sophisticated way for an application to be called at a given time is the timer chain. These are independent of event routines, but are used in a similar manner. OS_CallAfter (SWI &3B) can be used to get a

given address to be called after a certain time has elapsed. OS_CallEvery (SWI &3C) is like this, but automatically reloads the counter when it has expired. OS_RemoveTickerEvent (SWI &3D) will cancel either OS_CallAfter before it occurs or OS_CallEvery to stop it repeating forever..

OS_CallAfter and OS_CallEvery are passed an address to call, the delay to wait and an identification word to return in R12. Thus, many timers can be running concurrently.

These are stored in a list which can be any size up to the machine memory limit.

VSync timer

The screen is refreshed 50 times a second in Standard monitor type modes. From the time that the bottom of the screen is complete till the top of the screen commences again is a delay called the vertical sync period. This allows the electron beam to go to this start position. The VSync event coincides with the vertical sync beginning. You can use OS_Byte 14 to enable this event, so that flicker-free re-drawing can be done while the VDU is not being written to.

OS_Byte 176 provides access to a one byte counter in 50Hz periods. ie. it decrements at the rate of the VSync event.

Obsolete timers

OS_Byte 243 reads a temporary location used by the timer software. It is kept for compatibility with earlier Acorn operating systems and must not be used.

OS_Byte 176 (SWI &06)

Read/Write 50Hz counter

On entry

R0 = 176 (&B0) (reason code)
R1 = 0 to read or new value to write
R2 = 255 to read or 0 to write

On exit

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

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

This call reads or writes a one-byte counter which is decremented at a 50Hz rate; or more precisely at the rate of the VSync interrupt.

The write command can also be performed by `*FX 176,<value>`

Related SWIs

None

Related vectors

ByteV

OS_Byte 243 (SWI &06)

Read timer switch state

On entry

R0 = 243 (&F3) (reason code)
R1 = 0
R2 = 255

On exit

R0 = preserved
R1 = switch state
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

In order to protect the centi-second clock against corruption during reset, the OS keeps two copies. One of them is the one which will be read or written when one of the OS_Words is called, the other is the one which will be updated during the next 100Hz interrupt. When the update has been performed correctly, the values are swapped. This OS_Byte enables you to read the byte which indicates which copy is being used. Its only practical use is as a location which changes 100 times a second.

This call is obsolete and should not be used.

Related SWIs

OS_Word 3 (SWI &07), OS_Word 4 (SWI &07)

Related vectors

ByteV

OS_Word 1 (SWI &07)

Read system clock

On entry

R0 = 1 (reason code)
R1 = pointer to five byte block

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains the value of the system clock at the instant of the call.

R1+0 = time (least significant byte)
R1+1 = ...
R1+2 = ...
R1+3 = ...
R1+4 = time (most significant byte)

The clock is incremented every centi-second. The value of the clock is preserved over a soft break and set to zero after a hard break.

Related SWIs

OS_Word 2 (SWI &07)

Related vectors

WordV

OS_Word 2 (SWI &07)

Write system clock

On entry

R0 = 2 (reason code)
R1 = pointer to five byte block with centi-second clock value in it

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On entry, the parameter block contains the value to set the system clock.

R1+0 = time (least significant byte)
R1+1 = ...
R1+2 = ...
R1+3 = ...
R1+4 = time (most significant byte)

This allows the clock to be set to a specified value.

Related SWIs

OS_Word 1 (SWI &07)

Related vectors

WordV

OS_Word 3 (SWI &07)

Read interval timer

On entry

R0 = 3 (reason code)
R1 = pointer to five byte block

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains the value of the interval timer at the instant of the call.

R1+0 = time (least significant byte)
R1+1 = ...
R1+2 = ...
R1+3 = ...
R1+4 = time (most significant byte)

Like the system clock, the interval timer is incremented 100 times a second. The interval timer can be made to cause an event when its value reaches zero. To do this, it must be set to minus the number of centi-seconds that are to elapse before the event takes place.

To produce repeated events, the routine servicing the timer event should reload the timer with the appropriate number. For example, to produce an event every 10 seconds, reload it with –1000 (&FFFFFFFC18). An alternative is to use the special ticker event, described in the chapter entitled *Events*.

Related SWIs

OS_Word 4 (SWI &07)

Related vectors

WordV

OS_Word 4 (SWI &07)

Write interval timer

On entry

R0 = 4 (reason code)
R1 = pointer to five byte block

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On entry, the parameter block contains the value to set the interval timer.

R1+0 = time (least significant byte)
R1+1 = ...
R1+2 = ...
R1+3 = ...
R1+4 = time (most significant byte)

This call resets the interval timer to a specified value.

Note that you must use OS_Byte 14 to enable the interval timer event.

Related SWIs

OS_Word 3 (SWI &07)

Related vectors

WordV

OS_Word 14,0 (SWI &07)

Read soft-copy of the CMOS clock in string format

On entry

R0 = 14 (reason code)
R1 = pointer to parameter block
R1+0 = 0 (reason code)

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains a 25-byte character string in the form:

ddd,nn mmm yyyy.hh:mm:ss<Return> (starting from address R1)

where

ddd	is a three-character abbreviation for the day
nn	is the day number
mmm	is a three-character abbreviation for the month
yyyy	is the year
hh	is the hour (in 24-hr clock notation)
mm	is the number of minutes past the hour
ss	is the number of seconds

<return> a carriage return character (&0D).

This time string comes from the soft-copy of the 5 byte time maintained by RISC OS, not the CMOS clock chip itself.

This call is equivalent to the *Time command.

Related SWIs

OS_Word 15 (SWI &07)

Related vectors	WordV

OS_Word 14,1 (SWI &07)

Read CMOS clock in Binary Coded Decimal (BCD) format

On entry

R0 = 14 (reason code)
R1 = pointer to parameter block
R1+0 = 1 (reason code)

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On exit, the parameter block contains the seven-byte BCD clock value:

R1+0 = year	(00 - 99)
R1+1 = month	(01 - 12; 01 = January etc)
R1+2 = day of month	(01 - 31)
R1+3 = day of week	(01 - 07; 01 = Sunday etc)
R1+4 = hours	(00 - 23)
R1+5 = minutes	(00 - 59)
R1+6 = seconds	(00 - 59)

The clock value is read directly from the CMOS clock chip.

Related SWIs

OS_Word 15 (SWI &07)

Related vectors

WordV

OS_Word 14,2 (SWI &07)

Convert BCD clock value into string format

On entry

R0 = 14 (reason code)
R1 = pointer to parameter block

R1+0 = 2	reason code
R1+1 = year	(00 - 99)
R1+2 = month	(01 - 12; 01 = January etc)
R1+3 = day of month	(01 - 31)
R1+4 = day of week	(01 - 07; 01 = Sunday etc)
R1+5 = hours	(00 - 23)
R1+6 = minutes	(00 - 59)
R1+7 = seconds	(00 - 59)

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

On entry, the parameter block contains the 7-byte BCD clock value:

On exit, the parameter block contains a 25-byte character string in the form:

ddd,nn mmm yyyy.hh:mm:ss<Return> (starting from address R1)

where:

ddd	is a three-character abbreviation for the day
nn	is the day number
mmm	is a three-character abbreviation for the month
yyyy	is the year
hh	is the hour (in 24-hr clock notation)
mm	is the number of minutes past the hour
ss	is the number of seconds

<return> a carriage return character (&0D).

Related SWIs

OS_Word 15 (SWI &07)

Related vectors

WordV

OS_Word 14,3 (SWI &07)

Read real-time in 5-byte format

On entry

R0 = 14 (reason code)
R1 = pointer to parameter block
    R1+0 = 3 (reason code)

On exit

R0 = preserved
R1 = preserved
    R1+0 = LSB of time
    R1+1 = ...
    R1+2 = ...
    R1+3 = ...
    R1+4 = MSB of time

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The parameter block contains the 5-byte real time read from the soft copy of the system time clock. This number is in centi-seconds since 00:00:00 1st January 1900. It is used for time/date stamping by the filing system. It is also useful for utilities which are used for building consistent systems, eg 'Make'.

Related SWIs

OS_Word 15 (SWI &07)

Related vectors

WordV

OS_Word 15,8 (SWI &07)

Write the time only in the real-time and the CMOS clock settings

On entry

R0 = 15 (reason code)
R1 = pointer to parameter block
- R1+0 = 8 (reason code)
- R1+1 = ASCII code for first hours digit
- R1+2 = ASCII code for second hours digit
- R1+3 = 58 (ie ASCII code for :)
- R1+4 = ASCII code for first minutes digit
- R1+5 = ASCII code for second minutes digit
- R1+6 = 58
- R1+7 = ASCII code for first seconds digit
- R1+8 = ASCII code for second seconds digit

On exit

R0 = preserved
R1 = preserved

The C flag will be set on exit, if the parameter block contained a format error.

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

Change the time settings in the real-time value and the CMOS clock.

Related SWIs

OS_Word 14 (SWI &07)

Related vectors

WordV

OS_Word 15,15 (SWI &07)

Write the date only in the real-time and the CMOS clock settings

On entry

R0 = 15 (reason code)
R1 = pointer to parameter block
- R1+0 = 15 (reason code)
- R1+1 = ASCII code for first day character
- R1+2 = ASCII code for second day character
- R1+3 = ASCII code for third day character
- R1+4 = 44 (ie ASCII code for ',')
- R1+5 = ASCII code for first day digit
- R1+6 = ASCII code for second day digit
- R1+7 = 32 (ie ASCII code for space)
- R1+8 = ASCII code for first month character
- R1+9 = ASCII code for second month character
- R1+10 = ASCII code for third month character
- R1+11 = 32
- R1+12 = ASCII code for first year digit
- R1+13 = ASCII code for second year digit
- R1+14 = ASCII code for third year digit
- R1+15 = ASCII code for fourth year digit

On exit

R0 = preserved
R1 = preserved

The C flag will be set on exit, if the parameter block contained a format error.

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

Change the date settings in the real-time value and the CMOS clock.

Related SWIs	OS_Word 14 (SWI &07)
Related vectors	WordV

OS_Word 15,24 (SWI &07)

Write the time and date in the real-time and the CMOS clock settings

On entry

R0 = 15 (reason code)
R1 = pointer to parameter block
- R1+0 = 24 (reason code)
- R1+1 = ASCII code for first day character
- R1+2 = ASCII code for second day character
- R1+3 = ASCII code for third day character
- R1+4 = 44 (ie ASCII code for ',')
- R1+5 = ASCII code for first day digit
- R1+6 = ASCII code for second day digit
- R1+7 = 32 (ie ASCII code for space)
- R1+8 = ASCII code for first month character
- R1+9 = ASCII code for second month character
- R1+10 = ASCII code for third month character
- R1+11 = 32
- R1+12 = ASCII code for first year digit
- R1+13 = ASCII code for second year digit
- R1+14 = ASCII code for third year digit
- R1+15 = ASCII code for fourth year digit
- R1+16 = 46 (ie. ASCII code for period)
- R1+17 = ASCII code for first hour's digit
- R1+18 = ASCII code for second hour's digit
- R1+19 = 58 (ie ASCII code for :)
- R1+20 = ASCII code for first minute's digit
- R1+21 = ASCII code for second minute's digit
- R1+22 = 58
- R1+23 = ASCII code for first second's digit
- R1+24 = ASCII code for second second's digit

On exit

R0 = preserved
R1 = preserved

The C flag will be set on exit, if the parameter block contained a format error.

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

Change the time and date settings in the real-time value and the CMOS clock.

Related SWIs

OS_Word 14 (SWI &07)

Related vectors

WordV

OS_CallAfter (SWI &3B)

Call a specified address after a delay

On entry

R0 = time in centi-seconds
R1 = address to call
R2 = value of R12 to call code with

On exit

R0 = preserved
R1 = preserved
R2 = preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_CallAfter calls the code pointed to by R1 after the delay specified in R0. The code should regard itself as an interrupt routine, and behave accordingly.

OS_RemoveTickerEvent can be used to cancel a pending OS_CallAfter

Related SWIs

OS_CallEvery (SWI &3C), OS_RemoveTickerEvent (SWI &3D)

Related vectors

None

OS_CallEvery (SWI &3C)

Call a specified address every time a delay elapses

On entry

R0 = time in centi-seconds
R1 = address to call
R2 = value of R12 to call code with

On exit

R0 = preserved
R1 = preserved
R2 = preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_CallEvery calls the code pointed to by R1 every R0 centiseconds, until OS_RemoveTickerEvent is executed or Break is pressed. The code should regard itself as an interrupt routine, and behave accordingly.

Related SWIs

OS_CallAfter (SWI &3B), OS_RemoveTickerEvent (SWI &3D)

Related vectors

None

OS_RemoveTickerEvent (SWI &3D)

Remove a given call address and R12 value from the ticker event list

On entry

R0 = call address
R1 = value of R12 used in OS_CallEvery or OS_CallAfter

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_RemoveTickerEvent takes R0 as the address and R1 as the R12 value of the event to find and remove from its list.

It is used to stop an event set up by a call to OS_CallAfter or OS_CallEvery. The parameters passed must match those originally passed to OS_CallEvery or OS_CallAfter for it to remove the correct event.

Related SWIs

OS_CallAfter (SWI &3B), OS_CallEvery (SWI &3C)

Related vectors

None

OS_ReadMonotonicTime (SWI &42)

Number of centi-seconds since the last hard reset

On entry

—

On exit

R0 = time in centi-seconds

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadMonotonicTime returns the number of centi-seconds since the last hard reset, or switching on of the machine. 'Monotonic' refers to the fact that this timer is guaranteed to increase with time. It is used, for example, to time-stamp mouse events.

Related SWIs

None

Related vectors

None

OS_ConvertStandard DateAndTime (SWI &C0)

Convert 5-byte time into a string

On entry

R0 = pointer to 5-byte time block
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating zero in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ConvertStandardDateAndTime converts a five-byte value representing the number of centi-seconds since 00:00:00 on January 1st 1900 into a string. It converts it using a standard format string stored in the system variable 'SYS$DateFormat' and places it in a buffer (which should be at least 20 bytes).

See the *Technical Details* section of this chapter for details of the format field names.

Related SWIs

OS_ConvertDateAndTime (SWI &C1)

Related vectors

None

OS_ConvertDateAndTime (SWI &C1)

Convert 5-byte time into a string using a supplied format string

On entry

R0 = pointer to 5-byte time block
R1 = pointer to buffer for resulting string
R2 = size of buffer
R3 = pointer to format string (null terminated)

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating zero in buffer
R2 = number of free bytes in buffer
R3 = preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ConvertDateAndTime converts a five_byte value representing the number of centi-seconds since 00:00:00 on January 1st 1900 into a string. It converts it using the format string supplied.

Apart from the following exception, the format string is copied directly into the result buffer. However, whenever '%' appears in the format string, the next two characters are treated as a special field name which is replaced by a component of the current time.

See the *Technical Details* section of this chapter for details of the format field names.

Related SWIs

OS_ConvertStandardDateAndTime (SWI &C0)

Related vectors

None

*Time

Displays the day, date and time

Syntax

```
*Time
```

Parameters

—

Use

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

Related commands

None

Related SWIs

OS_Word 14,0 (SWI &07)

Related vectors

None

Conversions

Introduction

This chapter is a collection of SWIs that convert from one form to another. Here is a summary of the conversions that can be done:

- convert a number to a string in binary, decimal or hex, with some format control. You can specify that the source number in a variety of sizes. ie. 1, 2, 3 or 4 bytes in length in most cases.
- convert a string containing a number in any base from 2 to 36 to a number.
- process a string with control codes and other special characters. This allows a string with any control codes to be created by passing a string with only printable characters in it.
- substitute a string containing arguments with the given values. Used with command line arguments to an application.
- evaluate an expression with logical, arithmetic, bit and string operations, giving a logical, numeric or string result.
- given a key, extract options from a command line
- convert a SWI number to a string with its full name and vice versa.
- convert a network station pair of numbers number into a string.
- convert a file size into a string, for example "12 Kbytes"

Overview and Technical Details

This section leads through the details of the differing conversion calls. Whilst most are mutually independent, some SWIs may use others within this chapter to give a multi-layered functionality.

Numbers to strings

The simplest option to convert a signed 32-bit integer into a string, the most common operation, is to use OS_BinaryToDecimal (SWI &28).

For a far greater functionality, there is a set of 24 SWIs with a common calling convention that allow a wide ranging list of conversions. Generically, these SWIs are called OS_Convert<name><number> (SWI &D0 - E8). <name> refers to the destination format of the string. It can be hex, signed and unsigned integer (optionally with spaces between the thousands, millions and so on), or binary. The <number> is the number of bytes to use on input. For all apart from hex, this is 1, 2, 3, or 4 bytes. Hex can be 1, 2, 4, or 8 nibbles long. See the description of these SWIs for detail.

Note that OS_BinaryToDecimal is equivalent to OS_ConvertInteger4 (SWI &DC) from these SWIs.

Strings to numbers

OS_ReadUnsigned (SWI &21) will read a number in ASCII in a string and convert it into an unsigned integer. The number in the string can be specified to be in any base from 2 to 36. Base 36 has 0 - 9,A - Z as numbers. No prefix means that the number is decimal by default, while the conventional '&' is used to indicate hex. All bases can be specified by the base_number form. eg. 2_1100 is 12 in binary.

GS string operations

The GS operations are a way of putting any characters from 0 - 255 into a string using only the printable character set. OS_GSInit (SWI &25) and OS_GSRead (SWI &26) work together to scan a string on a character at a time basis. OS_GSTrans (SWI &27) performs both these functions and scans the string. Unless you need character by character control, OS_GSTrans is easier to use.

| character

The '|' character is used by the OS_GSRead and OS_GSTrans as a flag for a special character. It affects how the character following it is interpreted. Here is a list of its effects:

ASCII code	Symbols used
0	|@
1 - 26	|<letter> eg |A (or |a) = ASCII 1, |M (or |m) = ASCII 13
27	|[or |{
28	|\
29	|] or |}
30	|^ or |~
31	|_ or |'
32 - 126	keyboard character, except for:
"	|"
|	||
<	|<
127	|?
128 - 255	|!<coded symbol> eg ASCII 128 = |!|@ ASCII 129 = |!|A

Note that '|!' means set the top bit of the following character, even if it is set by another '|' character.

To include leading spaces in a definition, the string must be in quotation marks, "", which are not included in the definition. To include a single " character in the string, use |" or "".

Substitute arguments

The reason why '<' must be preceded by a '|' is that you can put values and variables inside angle brackets.

You can use the form <number>, where the number between the angle brackets will be interpreted as if it was a parameter to OS_ReadUnsigned. That is, a number in any base from 2 to 36. The value returned from this SWI will be placed as a character in the output stream. ie. any values above bit 7 will be ignored.

A string with a name enclosed in '< >' characters will be used to look up a system variable. You must have used *Set, *SetMacro or *SetEval to set the variable. The value of the variable will be substituted using OS_ReadVarVal for the name and the angle brackets. eg. if "hisname" had been set to "Fred",

will be used. ie. the same limitation as the numbers above. System variables and the calls that operate on them are described in the chapter entitled *Program Environment*.

Flags

There are options which can be used to determine the way in which the string is interpreted. This is done by setting the top three bits in R2 passed to OS_GSInit or OS_GSTrans, as follows:

Bit	Meaning
29	If set then a space is treated as a string terminator
30	If set control codes are not converted (ie '|' syntax is ignored)
31	Double quotation marks (") are not to be treated specially, ie they are not stripped around strings.

*Echo

The *Echo command will pass a string through OS_GSTrans and then send it to the display.

Evaluation operators

A string containing an expression can be evaluated. An expression consists of any of the operators listed below and strings and numbers. It can return a result that is a number or a string. OS_EvaluateExpression (SWI &2D) is the core routine here. It is in turn called by *Eval. This allows you to perform evaluations from the command line. *If uses this call to perform a logical decision about which * Command to perform.

Any strings in the evaluation string are passed to OS_GSTrans, so all its operators will be used. This of course means that OS_ReadUnsigned and OS_ReadVarVal will in turn be called if you use a string that requires them. Note, however, that vertical bar escape sequences (eg "\|G" for ASCII 7) are not recognised.

As well as passing <name> operators in strings to OS_ReadVarVal, any item which cannot immediately be treated as a string or a number is also looked up as a system variable. For example, in the expression FRED+1, FRED will be looked up as a variable.

The operators recognised by the expression evaluator are as follows:

Arithmetic operators

+	Add two integers
–	Subtract two integers
*	Multiply two integers
/	Integer part of division
MOD	Remainder of a division

Logical operators

=	Equal	–1 is TRUE
<>	Not equal	0 is FALSE
>=	Greater than or equal	
<=	Less than or equal	
<	Less than	
>	Greater than	

Bit operators

>>	Arithmetic shift right
>>>	Logical shift right
<<	Logical shift left
AND	AND
OR	OR
EOR	Exclusive OR
NOT	NOT

String operators

+	Concatenate two strings	eg "HI" + "LO" = "HILO"
RIGHT n	Take 'n' characters from the right.	eg "HELLO" RIGHT 2 = "LO"
LEFT n	Take 'n' characters from the left.	eg "HELLO" LEFT 3 = "HEL"
LEN	Return the length of a string	eg LEN "HELLO" = 5

Conversions

STR	Convert a number into a string	eg STR 24 = "24"
VAL	Take the value of a string	eg VAL "12d3" = 12

Where appropriate, type conversions are performed automatically. For example, if an integer is subtracted from a string, then the string is evaluated and an integer result is produced ("2"–1 gives the result 1). The null string "" is converted to 0 by both the implicit and explicit (VAL) conversions.

Similarly, integers will be converted to strings if necessary: the expression 1234 LEFT 2 will yield "12".

The operators have the same relative priorities as their equivalents in BBC BASIC, eg * is higher than + which is higher than >, etc.

Parameter substitution

Given a list of space separated arguments, OS_SubstituteArgs (SWI &43) will replace references to those parameters in a string. %0 refers to the first string in the argument list and so on. This is generally used when processing command lines.

For a more powerful handling of command lines, use OS_ReadArgs (SWI &49). This is passed a list of parameter definitions and an input string. The parameters can be described as being in any order or in a fixed order. They can handle on/off switches (ie. presence is indicated), or values. The values can also be automatically passed through OS_GSTrans or OS_EvaluateExpression if required.

SWI number to string

Two calls can be used to translate a SWI number to and from its full name as a string. OS_SWINumberToString (SWI &38) will go to a string and OS_SWINumberFromString (SWI &39) will convert from a string to a SWI number.

Note that having bit 17 set will result in the string being prefixed with an 'X' and vice versa.

Econet numbers

The pair of numbers that refer to the network number and station number can be converted into a string by OS_ConvertFixedNetStation (SWI &E9). This will pad the string with leading zeros where required. If you don't want this padding, then OS_ConvertNetStation (SWI &EA) will do this.

File size

There are two SWIs that will convert a file size from an integer into a string. They can decide whether to display as bytes, Kbytes or Mbytes. OS_ConvertFileSize (SWI &EC) will convert an integer into a number up to 4 digits followed by an optional 'K' if it is in kilobytes or 'M' if in megabytes, followed by the word "bytes" and a null to terminate.

OS_ConvertFixedFileSize (SWI &EB) is exactly the same, except that it will always print the numeric field as fours characters, padding with spaces if necessary.

OS_ReadUnsigned (SWI &21)

Convert a string to an unsigned number

On entry

R0 = base in the range 2 - 36 (else 10 assumed), and flags in top 3 bits
R1 = pointer to string
R2 = maximum value if R0 bit 29 set

On exit

R0 preserved
R1 = pointer to terminator character
R2 = value

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadUnsigned takes a pointer to a string and tries to convert it into an integer value which is returned in R2.

Valid strings may start with a digit (where 'digits' may also be letters, depending on the base) or one of the following:

&	The number is in hexadecimal notation
base_	The number is in a given base, where 'base' is in the range 2 to 36. For example, 2_1010 is a base two (binary) number.

These override any base specified in R0. (If R0 contains an illegal base, 10 is assumed.) Characters following them are read until a character is reached which is not consistent with the base in use. For example, assuming R0=10 on entry, the terminator of 43AZ is A, whereas the terminator of &43AZ is Z.

In addition, R0 contains three flags which cause checks to be performed on the terminator and the range of the number obtained:

Bit	Meaning if set
31	Check terminator is a control character, space
30	Restrict value range to 0 - 255
29	Restrict range to 0 - R2 inclusive; a `Number too big` error is given otherwise

If either of these checks fail, a `Bad number` error is given. This error also occurs if the first character is not a valid digit. If a base is given at the start of the number and isn't in the range 2 - 36, a `Bad base` error is given.

Related SWIs

None

Related vectors

None

OS_GSInit (SWI &25)

Initialises registers for use by OS_GSRead

On entry

R0 = pointer to string to translate
R2 = flags

On exit

R0 = value to pass back in to OS_GSRead
R1 = first non-blank character
R2 = value to pass back in to OS_GSRead

Interrupts

Interrupt state is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_GSInit is one of the string routines which are used by the operating system command line interpreter to process the strings sent to it. One of the advantages of these routines is that they enable you to use the character '|' to introduce control characters which would otherwise be difficult to enter directly from the keyboard.

See the *Technical details* section of this chapter for a list of the conversions that are performed by the routines and the flags passed in R2.

OS_GSInit also returns the first non-blank character in the string. However, this is not necessarily the same as the output from the first OS_GSRead since OS_GSInit doesn't perform any expansion.

Related SWIs

OS_GSRead (SWI &26), OS_GSTrans (SWI &27)

Related vectors

None

OS_GSRead (SWI &26)

Returns a character from a string which has been initialised by OS_GSInit

On entry

R0 from last OS_GSRead/OS_GSInit
R2 from last OS_GSRead/OS_GSInit

On exit

R0 = updated
R1 = next translated character
R2 = updated
C flag is set if end of string reached

Interrupts

Interrupt state is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_GSRead reads a character from a string, using registers initialised by a OS_GSInit immediately prior to this call. The next expanded character is returned in R1. The values in R0 and R2 are updated so they are set up for the next call to OS_GSRead.

The interpretation of characters which pass through OS_GSRead is described in the *Technical details* section of this chapter.

An error is returned for a bad string – for example, mismatched quotation marks.

Related SWIs

OS_GSInit (SWI &25), OS_GSTrans (SWI &27)

Related vectors

None

OS_GSTrans (SWI &27)

Equivalent to a call to OS_GSInit and repeated calls to OS_GSRead

On entry

R0 = string pointer, terminated by 10 or 13 or 0
R1 = buffer pointer
R2 = buffer size (maxlen) and flags in top 3 bits

On exit

R0 = pointer to character after terminator
R1 = pointer to buffer, or 0
R2 = number of characters or maxlen+1 if it overflowed
C flag is set if buffer overflowed

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_GSTrans is equivalent to a call to OS_GSInit followed by repeated calls to OS_GSRead until the end of the source string is reached. Each time it obtains a character and translates it, OS_GSTrans then places it in a buffer.

The flags in R2, on entry, are the same as those supplied to OS_GSInit. On exit, R0 points to the character after the terminator of the source string, and R1+R2 points to the terminator of the translated string. If C=1 on exit, R2 is set to the length of the translated string buffer plus one.

The flags and interpretation of characters which pass through OS_GSTrans are described in the *Technical details* section of this chapter.

An error is returned for a bad string – for example, mismatched quotation marks.

Related SWIs

OS_GSInit (SWI &25), OS_GSRead (SWI &26)

Related vectors

None

OS_BinaryToDecimal (SWI &28)

Convert a signed number to a string

On entry

R0 = signed 32-bit integer
R1 = pointer to buffer
R2 = maximum length

On exit

R0, R1 preserved
R2 = number of characters given

Interrupts

Interrupt state is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_BinaryToDecimal takes a signed 32-bit integer in R0 and converts it to a string, placing it in the buffer. R1 points to the buffer and R2 contains its maximum length. Leading zeros are suppressed and the string will start with a minus sign, '–', if R0 was negative. The number of characters given is returned in R2.

The error `Buffer overflow` is given if the converted string is too long to fit in the buffer. An error is also given for a bad string – for example, mismatched quotation marks.

Related SWIs

None

Related vectors

None

OS_EvaluateExpression (SWI &2D)

Evaluate a string expression and return an integer or string result

On entry

R0 = pointer to string
R1 = pointer to buffer
R2 = length of buffer

On exit

R0 preserved
R1 = 0 if an integer returned, else preserved
R2 = integer result, or length of string in buffer

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_EvaluateExpression takes a string pointed to by R0, evaluates it and places the result in the buffer which is pointed to by R1. Its maximum length is R2. The type of the result is given by R1 as follows:

Value	Meaning
0	Integer result returned in R2
Not 0	String is returned in buffer, length returned in R2, R0 and R1 preserved

If the buffer is not large enough to hold the resulting string, then a `Buffer overflow` error is generated.

See the description in the *Technical details* of the operators that you can use.

Related SWIs

None

Related vectors

None

OS_SWINumberToString (SWI &38)

Convert a SWI number to a string containing its name

On entry

R0 = SWI number
R1 = pointer to buffer
R2 = buffer length

On exit

R0, R1 preserved
R2 = length of string in buffer

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_SWINumberToString converts a SWI number to a SWI name.

The returned string is null-terminated, and starts with an X if the SWI number has bit 17 set.

SWI numbers < &200 have an 'OS_' prefix to the main part, and a SWI-dependent end section (which is 'Undefined' for unknown OS SWIs).

SWI numbers in the range &100 to &1FF are converted in the form OS_Write+"A", or OS_WriteI+23 if the character is not a printable one.

SWI numbers &200 are looked for in modules. If a suitable name is found, it is given in the form module_name or module_number, eg. Wimp_Initialise, Wimp_32. If no name is found in the modules, the string 'User' is returned.

Related SWIs

OS_SWINumberFromString (SWI &39)

Related vectors

None

OS_SWINumberFromString (SWI &39)

Convert a string to a SWI number if valid

On entry

R1 = pointer to name (which is terminated by a character ≤ 32)

On exit

R0 = SWI number
R1 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_SWINumberFromString converts a SWI name to a SWI number. An error is given if the SWI name is not recognized.

The conversion is as follows:

- A leading X is checked for and stripped. If present, it will cause &20000 to be added to the number returned. (Bit 17 will be set.)
- System names are checked for. Note that the conversion of SWIs is not quite bidirectional: the name OS_WriteI+" " can be produced, but only OS_WriteI is recognized.
- Modules are scanned. If the module prefix matches the one given, and the suffix to the name is a number, then that number is added to the module's SWI 'chunk' base, and the sum returned. For example, Wimp_&23 returns &400E3, as the Wimp's chunk number is &400C0.
- If the suffix is a name, and this can be matched by the module, the appropriate number is returned. For example, Wimp_Poll returns &400C7.

See the chapter entitled *Modules* for more information on how modules provide the conversion.

Note that SWI names are case sensitive, so you must spell them exactly as returned by OS_SWINumberToString.

Related SWIs

OS_SWINumberToString (SWI &38)

Related vectors

None

OS_SubstituteArgs (SWI &43)

Substitute command line arguments

On entry

R0 = pointer to argument list, and flag in top bit
R1 = pointer to buffer for result string
R2 = length of buffer
R3 = pointer to template string
R4 = length of template string

On exit

R0, R1 preserved
R2 = number of characters in result string (inc. terminator)
R3, R4 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call performs the hard work involved in substituting a list of arguments into a 'template' string. Its main use is in the processing of command Alias$ variables by the system. As it is also useful in other situations, it has been made available to users. For example, FileSwitch uses it in the processing of Alias$@LoadType_TTT variables.

The argument list is a string consisting of space-separated items which will be substituted into the template string. Spaces within double quotation marks are not counted as argument separators. Typically, the argument string will just be the tail of a * Command. It is control-character terminated.

The result of substituting the arguments into the template string is placed in the buffer. The length of the buffer is given so that the call can check for buffer overflow.

The template string is copied into the result buffer character for character. However, when a '%' appears in the template string (even within quotation marks), it marks where an argument should be placed into the output buffer.

The '%' is followed by a single digit from 0 to 9. %0 stands for the first argument in the argument list, and so on. %*n means all of the arguments from number n onwards. %% means a single '%'. Anything else following the '%' is not treated specially, ie both the '%' and the character are copied over.

The template string does not have a terminator; instead its length is given. At the end of the substitution, any arguments after the highest one mentioned in the template string are appended to the result string. This can be stopped by setting the top bit of R0 on entry.

If a non-existent argument is specified in the template string, then the substitution process is terminated. No error is given.

Related SWIs

None

Related vectors

None

OS_ReadArgs (SWI &49)

Given a keyword definition, scan a command string

On entry

R0 = pointer to keyword definition
R1 = pointer to input string
R2 = pointer to output buffer
R3 = size of output buffer

On exit

R0 - R2 preserved
R3 = bytes left in output buffer

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI processes a command string using a keyword definition for syntax. The results are written out to the output buffer using a specialised format for this command.

Keyword definition

The keyword definition defines the parameters that can be in the command string. It is composed of a sequence of keywords, separated by commas. Each of these is made up of one or two names, followed by a sequence of qualifiers. The syntax of a keyword is:

[<keyword_name>[=<alias_name>]][/<qualifier>...]

The keyword_name is what you want users to identify the parameter with. This can be any string composed of alphanumerics and the '_' character. The alias name is an optional alternative name for the same keyword. You can have a keyword with no name. See the command string description below of how to set it.

The qualifier describes what kind of a parameter it is. There can be as many qualifiers as you like with one parameter, but some are mutually exclusive. The qualifiers can be any one of the following characters in upper or lower case:

- /A – keyword must always be given a value
- /K – keyword must always precede its value
- /S – the option is a switch. ie. presence only is reported
- /E – OS_EvaluateExpression will be called to transform the value. This can return a number or a string. Note that numeric evaluations only can be performed.
- /G – OS_GSTrans will be called to transform the value

Command string

The command string contains a sequence of commands using the syntax defined by the keyword definition. A command string is made of definitions of the following syntax:

[–<keyword_name>] <value>

If the keyword name is used, then the value will be attached to the named keyword. These can appear in any arbitrary order in the command string. The name after the '–' can be the full name of the keyword or its alias, or the first letter of either. For example, if the keyword definition contains "name=title", then all of the following are valid in the command string:

"–name fred", "–title fred", "–n fred", "–t fred"

Note that if more than one keyword has the same first letter, then the single letter form will be used by the first occurrance of a given letter in the keyword definition.

Also note that case is ignored, so "–FILE" and "–file" are identical.

If a definition has no *–keyword_name* preceding it, then the first unused keyword that is not a switch in the definition string will be given that value. This is how nameless keywords are set. For example, if the definition string is "infile,/a,outfile" and the command string is "–infile one –outfile two three", then the first and nameless keyword will be set to three, because it was the first undefined keyword in the definition.

Keywords are marked by a preceding '–' character, but this does not disallow these characters from appearing in values anywhere but at the start For example, if the keyword definition is "formula/e", then "–formula 6–3" will set it to the value of 3. If the command is "–formula –3+6", then this will cause an error.

Whilst some evaluated expressions can be done without spaces (1+2 for example), there are many that cannot. You can evaluate an expression in quotes, which allow spaces, as in this example:

```
"&3F AND &17"
```

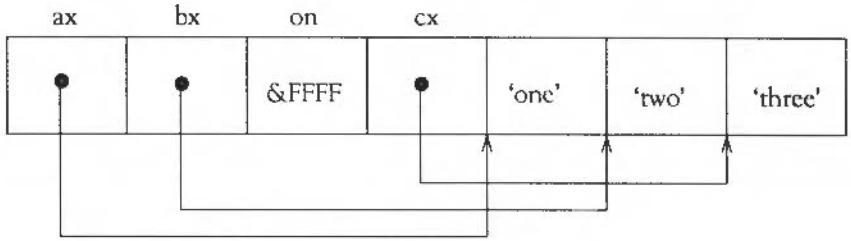
With GSTransed strings, if you want to put a quoted string inside quotes then you must use double quotes, as follows:

```
"This is ""IT"""
```

Output buffer

The output buffer contains the results for all of the possible keywords. For N keywords in the keyword definition, the first N words of the output buffer contain the results of the parsing of the command line. If the keyword was a switch (with /S qualifier), then a non-zero value indicates that the switch was used. For all other kinds of result, then it is a pointer. These results are appended sequentially to the output buffer.

The following example uses a keyword definition of "ax,bx,on/s,cx" and a command string of "one two three –on". The output buffer looks like this:



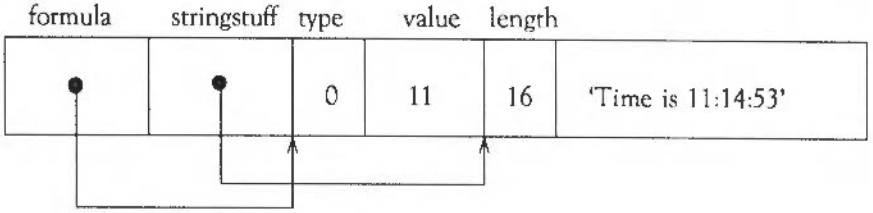
The results of GSTransed strings and evaluated expressions are stored differently. In a GSTransed string, the result pointer points to a block of the following format:

length	two byte length
string	length bytes of string

In an evaluated expression, the pointer points to a block like the following:

type	one byte result type. At present, this can only be zero, an integer
value	four byte integer

For an example showing /e and /g switches, if the keyword definition was "formula/e,time/g" and the command string was "–f 6+6–1 –t ""Time is <Sys$Time>""", then the result looks like this:



Examples

Keyword definition: number=times/e,file/k/a,expandtabs/s

can be matched by:

–n 10 –file jeff
–times 1+7 –file jeff –expandtabs
–file thingy –e

but not by:

thingy –number 4
–number 20 –times 4 –file jeff

Related SWIs

None

Related vectors

None

OS_Convert<name><number> (SWIs &D0 - E8)

These calls convert a number into a string

On entry

R0 = value to be converted
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWIs are re-entrant

Use

This range of SWIs use a common form and can convert a number into a string in a variety of ways.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

The <name> part of the SWI name can be any of the following groups:

Hex

Convert to a hexadecimal string.

The <number> is the number of ASCII digits in the output string, either 1, 2, 4, 6 or 8. Only enough significant bits to perform the conversion are used, and leading zeros are always included, so the string is fixed length. No ampersand ('&') is included in the string. The SWIs used in this group are:

SWI name	SWI No.	Output for... zero	largest val.
OS_ConvertHex1	&D0	'0'	'F'
OS_ConvertHex2	&D1	'00'	'FF'
OS_ConvertHex4	&D2	'0000'	'FFFF'
OS_ConvertHex6	&D3	'000000'	'FFFFFF'
OS_ConvertHex8	&D4	'00000000'	'FFFFFFFF'

Cardinal

Convert to an unsigned decimal number.

The <number> is the number of bytes to be used from the input value. The string is not padded with zeros, so is of variable length. The SWIs used in this group are:

SWI name	SWI No.	Output for... zero	largest value
OS_ConvertCardinal1	&D5	'0'	'255'
OS_ConvertCardinal2	&D6	'0'	'65535'
OS_ConvertCardinal3	&D7	'0'	'16777215'
OS_ConvertCardinal4	&D8	'0'	'4294967295'

Integer

Convert to a signed decimal number.

The <number> is the number of bytes to be used from the input value. The string is not padded with zeros, so is of variable length. If the most significant bit of the N bytes used is set, the number is taken to be negative, and a leading '–' is produced. The SWIs used in this group are:

SWI name	SWI No.	Output for... largest –ve	largest +ve value
OS_ConvertInteger1	&D9	'–128'	'127'
OS_ConvertInteger2	&DA	'–32768'	'32767'
OS_ConvertInteger3	&DB	'–8388608'	'8388607'
OS_ConvertInteger4	&DC	'–2147483648'	'2147483647'

Binary

Convert to a binary number.

The <number> is the number of bytes to be used from the input value. The string is padded with leading zeros, so the length is N*8. The SWIs used in this group are:

SWI name	SWI No.	Output for largest value
OS_ConvertBinary1	&DD	'11111111'
OS_ConvertBinary2	&DE	'1111111111111111'
OS_ConvertBinary3	&DF	'111111111111111111111111'
OS_ConvertBinary4	&E0	'11111111111111111111111111111111'

SpacedCardinal

Convert to an unsigned decimal number, with spaces every three digits.

The <number> is the number of bytes to be used from the input value. The string is not padded with zeros, so is of variable length. In addition, every three digits from the right, a space is inserted. The SWIs used in this group are:

SWI name	SWI No.	Output for... zero	largest value
OS_ConvertSpacedCardinal1	&E1	'0'	'255'
OS_ConvertSpacedCardinal2	&E2	'0'	'65 535'
OS_ConvertSpacedCardinal3	&E3	'0'	'16 777 215'
OS_ConvertSpacedCardinal4	&E4	'0'	'4 294 967 295'

SpacedInteger

Convert to a signed decimal number.

The <number> is the number of bytes to be used from the input value. The string is not padded with zeros, so is of variable length. If the most significant bit of the N bytes used is set, the number is taken to be negative, and a leading '–' is produced. The SWIs used in this group are:

SWI name	SWI No.	Output for... largest –ve	largest +ve val.
OS_ConvertSpacedInteger1	&D9	'–128'	'127'
OS_ConvertSpacedInteger2	&DA	'–32 768'	'32 767'
OS_ConvertSpacedInteger3	&DB	'–8 388 608'	'8 388 607'
OS_ConvertSpacedInteger4	&DC	'–2 147 483 648'	'2 147 483 647'

Related SWIs	OS_BinaryToDecimal (SWI &28)
Related vectors	None

OS_ConvertFixedNetStation (SWI &E9)

Convert from an Econet station/network number pair to a string

On entry

R0 = pointer to two word block (value to be converted)
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null zero in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

R0 points to two words in memory. The first word contains the station number and the second word contains the network number.

This call always converts into a form nnn.sss, where nnn is the network number. If it is zero, the first four characters are spaces. If it is non-zero, leading zeros are converted to spaces. sss is the station number. If the network was zero, leading zeros in the station number are converted to spaces, otherwise they are left as zeros.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertNetStation (SWI &EA)

Related vectors

None

OS_ConvertNetStation (SWI &EA)

Convert from an Econet station/network number pair to a string

On entry

R0 = pointer to two word block (value to be converted)
R1 = pointer to buffer for resulting string
R2 = size of buffer

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

R0 points to two words in memory. The first word contains the station number and the second word contains the network number.

This call performs the same conversion as OS_ConvertFixedNetStation, but suppresses zeros and spaces wherever possible, to yield the shortest possible string.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertFixedNetStation (SWI &E9)

Related vectors

None

OS_ConvertFixedFileSize (SWI &EB)

Convert an integer into a filesize string of a fixed length

On entry

R0 = filesize in bytes
R1 = pointer to buffer
R2 = length of buffer in bytes

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI will convert an integer into a filesize string of a fixed length. The format of the string is:

<4 digit number><space><space | K | M>"bytes"<null>

The four digit number at the start is padded with spaces if there aren't enough in the number.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertFileSize (SWI &EC)

Related vectors

None

OS_ConvertFileSize (SWI &EC)

Convert an integer into a filesize string

On entry

R0 = filesize in bytes
R1 = pointer to buffer
R2 = length of buffer in bytes

On exit

R0 = pointer to buffer (R1 on entry)
R1 = pointer to terminating null in buffer
R2 = number of free bytes in buffer

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI will convert an integer into a filesize string. The format of the string is:

<number><space><space | K | M>"bytes"<null>

The number at the start is up to four digits in length.

R0 returns pointing to the start of the buffer. This is convenient for calling OS_Write0. R1 points to the null at the end of the buffer. This is convenient for adding further text after it.

Related SWIs

OS_ConvertFixedFileSize (SWI &ED)

Related vectors

None

*Echo

Display a string on the screen

Syntax

```
*Echo <string>
```

Parameters

`<string>` string to display

Use

*Echo takes the string following it, translates it using OS_GSTrans and then displays it on the screen.

Example

```
*Echo |GError!|M
```

Related commands

None

Related SWIs

None

Related vectors

None

*Eval

Evaluates an integer, logical or string expression

Syntax

```
*Eval <expression>
```

Parameters

`<expression>` any combination of the operations listed below

Use

*Eval evaluates an integer, logical, bit or string expression, carrying out type conversions where necessary, in a similar way to the BASIC EVAL command. It will not handle floating point numbers. You can use *Eval to do simple arithmetic (although the desktop Calculator is easier to use for four-function arithmetic), or to evaluate more complex expressions. Programmers may find the command useful for doing 'offline' calculations (checking on space left, for example).

See the description in the technical description of the operators that you can use.

Example

```
*Eval 127 * 23 >> 2
```

Related commands

*If, *SetEval

Related SWIs

OS_EvaluateExpression (SWI &2D)

Related vectors

None

*If

Allows you to execute * Commands conditionally

Syntax

```
*If <expression> Then <command> [Else <command>]
```

Parameters

<expression>	an integer expression
<command>	any valid * Command

Use

The *If command allows you to execute * Commands conditionally. <expression> can be any integer expression, including variable names enclosed in angled brackets. The expression is evaluated by the operating system's expression evaluation.If the If-expression evaluates to a non-zero value, the Then-clause is executed. If the If-expression evaluates to zero, and there is an Else-clause, the Else-clause is evaluated.

If you wish to compare a variable to a string both must be enclosed in double quotes to ensure a string comparison is performed; see the first example.

See the description in the technical description of the operators that you can use.

Example

```
*If "<name>" = "Zaphod" Then echo Hello Zaphod! Else
Echo Go away <name>!

If <Sys$Year>=1988 Then Run Calendar
```

Related commands

*Eval

Related SWIs

None

Related vectors

None

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_Init 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*, 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 }
```

In the third example, `out1` will be created with nothing in it, input will be read from `infile` and output will go to `out2`.

The final example shows how redirections can be concatenated within the same pair of braces.

Aliases

An alias is a variable of the form Alias$cmd, 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 Alias$SetPS, and that this has the value –NET–PS %0|MConfigure PS %0. The OS will match the command name against the alias variable. It will then substitute all occurrences of %0 in the variable's value by 0.235. Then, the two lines of the variable will be executed thus:

```
-NET-PS 0.235
Configure PS 0.235
```

So, the net effect of executing the original command is to set the network printer server both temporarily, and also in the permanent configuration.

Another example using the parameter substitution is

```
*Set Alias$Mode Echo |<22> |<%0>
```

The '|'s before the angle brackets are to stop them from being evaluated when the *Set command is entered. Typing *Mode n will then set the display to mode 'n'.

Look-up the command

After all the previous steps have been completed, the command that is left after pre-processing must be executed. This is a list in order of the things that RISC OS will check to execute a command:

- 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* 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* 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.

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.

The command is then executed as any other * Command. This is described in the technical description.

Related SWIs None

Related vectors CLIV

*GOS

Calls Command Line Mode and 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 un-saved work.

See the technical description section in this chapter for a description of how the command line interface works.

Related commands

*Quit, *Desktop

Related SWIs

None

Related vectors

None

Modules

Introduction

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

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

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

Several SWIs and * Commands are provided by the operating system for handling modules. For example, loading a module file from the filing system.

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

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

This chapter describes what is needed to write a module.

Overview

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

Using modules

Use of modules is centralised around the SWI OS_Module. This contains 14 operations that can:

- Load, initialise, run and remove a module
- Examine and change the amount of RMA space used by a module
- Examine module details
- Modify instantiations of modules

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

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

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

Instantiation

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

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

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

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

```
*Modules
...
6  03839698  01803FE4  FileCore%RAM
   03839698  01800F34  FileCore%ADFS
   03839698  01803FE4  FileCore%Base
...
```

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

```
*RMKill WaveSynth%Base
```

Writing a module

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

Module header

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

- Where to start executing in the module. This is used by languages and applications.
- Where to call initialisation code. This has to be called before all the others.
- Where to call finalisation code. This is called before removing the module. It allows the module to shutdown any hardware it is using and generally tidy up.
- A title for the module
- A help string. This is used automatically by RISC OS when help is requested.

- Detailed help on * Commands
- Entry points for * Commands. RISC OS will decode the * Commands and call the right entry point for a command for you.
- A table to convert to and from SWI names and numbers
- Entry points for all the SWIs in the module
- The chunk number for the module. This is the number that is the base for SWI numbers. There can be up to 64 SWIs in a module, all offsets from this chunk number. This is the only entry in the header that isn't a pointer.
- Service call entry (see below)

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

Service calls

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

- Unknown command, OS_Byte, OS_Word, *Configure or *Status.
- *Help has been called. This allows you to replace this command when you detect a particular help call being made.
- Memory controller about to be remapped. This allows an application to stop a memory remapping if it doesn't want it to happen.
- Application is about to start. This allows a module to prevent an application from starting. With this, a module could prevent any other tasks running.
- Lookup file type. This converts the 3 byte file type into a string, such as 'BASIC' or 'Text'.
- Various international services, such as handling different alphabets and keyboards.

- The fast interrupt handler has been claimed/released. This is used by device drivers for high data rate devices that depend on the state of the fast interrupt system.

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

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

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

Technical Details

Using modules

OS_Module (SWI &1E) is the main application interface to modules. In its description you will find a complete list of its calls and details of each.

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

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

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

*RMEnsure is a command that will check that a given module and version number is loaded into memory and will try and load it if it is not.

*UnPlug will disable the ROM version of a given module. This is used if an upgraded version of a module is released and can be loaded from a filing system.

Workspace

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

The system works on the assumption that the private word is a pointer to workspace claimed in the RMA. It therefore provides suitable default actions on that basis. For example, the system will attempt to free any workspace claimed using this pointer.

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

Note that workspace allocated through XOS_Module will always lie on an address &XXXXXX4. This enables code written for time-critical software (eg sound voice generators and FIQ handlers) to be aligned within the module body.

Errors in module code

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

- Set up R0 to point to the error block
- Preserve all appropriate registers
- Return with V set.

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

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

Module header format

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

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

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

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

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

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

Service calls

Service calls are made from RISC OS to a module to indicate an occurrence of some kind. Some are claimable, and some are intended as broadcasts of the occurrence only. See the description in OS_Service (SWI &30) for a complete list of all service calls. It is followed by details of each call. Some of these service calls will also be relevant to other parts of this manual that describe modules. For example, there are service calls that are provided explicitly to serve the International module.

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

Module entry points

Start code

Start executing at the start point of code in a module

Offset in header

&00

On entry

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

On exit

Doesn't return unless error occurs.

Interrupts

Interrupts are enabled on entry
Fast interrupts are enabled

Processor Mode

Processor is in USR mode

Re-entrancy

Entry point is not re-entrant

Use

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

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

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

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

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

ıitialisation Code	Set up the module, so that all other entry points are operating
Offset in header	&04
Jn entry	R10 = pointer to environment string (ie initialisation parameters supplied by caller of OS_Module) R11 = I/O base or instantiation number R12 = pointer to currently preferred instantiation of the module. If the word ≠ 0, this implies reinitialisation R13 = supervisor stack
On exit	Must preserve processor mode and interrupt state Must preserve R7 - R11 and R13 R0 - R6, R12, R14 and the flags (except V of course) can be corrupted
Interrupts	Interrupts are enabled Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
ıe-entrancy	Entry point is not re-entrant
Use	This code is called when the module is loaded and also after the RMA has been tidied (OS_Module with Tidy reason code). It is defined that the module will not be called via any other entry point until this entry point has been called. Thus the initialisation code is expected to set up enough information to make all other entry points safe. An offset of zero means that the module does not need any initialisation. The system does not provide any default actions. The Initialisation code is used by OS_Module with Run, Load, ReInit and Tidy reason codes. If the module is being re-entered after a OS_Module 'tidy', the private word may contain a non-zero value. This is the contents of the private word before the finalisation, relocated (if necessary) by the system. Typical actions are claiming workspace (via OS_Module) and storing the workspace pointer in the private word. Other actions may include linking onto vectors, declaring the module as a filing system, etc.

The module can refuse to be initialised. If an error is generated during initialisation, the system removes the module and any workspace pointed to by its private word from the RMA. Any error should be dealt with by setting R0 to be an error indicator and returning to the module handler with V set.

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

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

On exit, use the link register passed in R14 to return:

```
MOV PC,R14
```

Return V set or clear depending on whether an error has occurred or not. If an error has occurred, it returns R0 as the error indicator.

Finalisation Code

Called before killing the module

Offset in header

&08

On entry

R10 = fatality indication. 0 is non-fatal, 1 is fatal
R11 = instantiation number
R12 = pointer to currently preferred instantiation of the module.
R13 = supervisor stack

On exit

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

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

This is the reverse of initialisation. This code is called when the system is about to kill all instantiations of the module either completely or temporarily whilst it tidies the RMA.

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

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

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

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

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

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

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

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

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

Service call handler

Called when a service call is issued

Offset in header

&0C

On entry

R1 = service number
R12 = pointer to currently preferred instantiation of the module
R13 = a full, descending stack

On exit

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

Interrupts

Interrupts are undefined on entry
Fast interrupts are enabled

Processor Mode

Processor is in SVC or IRQ mode

Re-entrancy

Entry point is not re-entrant

Use

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

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

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

The service call handler is used when a service call is issued or via an OS_Byte 143 (SWI &06) or OS_ServiceCall (SWI &30). The service calls are described in the section on OS_Service.

Title string

Offset of a null-terminated module name

Offset in header

&10

Use

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

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

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

Used by OS_Module with reason codes Delete, Enter and ReInit. Also printed by the *Modules command.

Help string

Used when *Help prints information from the module

Offset in header

&14

Use

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

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

Module name <Tab>[<Tab>] v.vv (DD MMM YYYY)

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

Help and command keyword table

Get help on * Commands or enter them

Offset in header

&18

On entry

R12 = pointer to currently preferred instantiation of the module
R13 = pointer to a full descending stack
R14 = return address

On exit

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

Interrupts

Interrupts are enabled on entry
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

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

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

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

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

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

The code offset is used for commands. A zero entry means that the string has help text only associated with it. The code is entered with R0 pointing at the command tail and R1 set to the number of parameters (as counted by OSCLI, which means space(s) separate parameters except within double quotation marks).

Information word

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

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

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

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

The flags are as follows:

Bit 31 = 1

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

Bit 30 = 1

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

R0 = 0 *Configure with no option has been received
The module prints a syntax string and return.
R0 = 1 *Status <keyword> has been issued. The module should print the currently configured status for this keyword.

If R0 is neither of the above, it means that the *Configure <option> has matched <option> against the keyword and R0 is a pointer to the command tail with leading spaces skipped. The arguments are decoded and the configuration set accordingly. If the command tail is incorrect, the module should return with V set and R0 indicating the error as follows:

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

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

Bit 29 = 1

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

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

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

Other comments

Other flags should be zero for upwards compatibility. The invalid syntax message is used by OSCLI as the text of an error message. If the parameters, which are given, fall outside the range specified. If a zero offset is given, a default `Invalid number of parameters` error is given instead.

The help text is used by *Help. If a keyword in the *Help command tail matches the match string, then the help text is pretty-printed using the RISC OS internal token dictionary. Refer to OS_PrettyPrint (SWI &44) for a full list of the token dictionary.

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

SWI chunk base number

The base of chunk numbers for the module

Offset in header

&1C

Use

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

SWI handler code

Called to handle SWIs belonging to the module

Offset in header

&20

On entry

R11 = SWI number modulo Chunk Size (ie 0 - 63)
R12 = private word pointer
R13 = supervisor stack
R14 contains the flags of the SWI caller

On exit

R10 - R12 may be corrupted
Use `MOVS PC,R14` to return, having altered R14 flags as appropriate
(eg. setting V for an error).

Interrupts

Interrupts are disabled on entry
Fast interrupts are enabled

Interrupts should always be enabled if SWI processing will take a long time (say > 20us) and the routine can cope with IRQs being enabled. The code to enable IRQs is:

```
MVN   Rn, #I_bit; = &08000000
TSTP  Rn, PC      ; preserves other flags
```

To disable IRQs explicitly:

```
MOV   Rn, PC
ORR   Rn, #I_bit
TEQP  Rn, #0
```

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

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

Base to base + (SWI chunk size – 1)

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

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

- Base is a multiple of the chunk size and has a 0 top byte

- Code offset is a multiple of four with the top six bits zero

See the introductory section of this manual for more details on how to choose a chunk number.

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

```
.SWIentry
      LDR   R12, [R12]          ; get workspace pointer
      CMP   R11, #(EndOfJumpTable - JumpTable)/4
      ADDCC PC, PC, R11, LSL #2 ; dispatch if in range
      B     UnknownSWIerror     ; unknown SWI
.JumpTable
      B     MySWI_0
      B     MySWI_1
      ..............
      B     MySWI_n
.EndOfJumpTable
.UnknownSWIError
      ADR   R0, errMesg
      ORRS  PC, R14, #Overflow_Flag
.errMesg
      EQUD  &1E6                ;Same as system message
      EQUS  "Unknown <module> operation"
      EQUB  0
```

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

The R14 given to the SWI code contains the flags of the SWI caller, except that V has been cleared. So, to return without updating the flags, use `MOVS PC, R14`. Otherwise alter the link register (for example by executing `ORRS PC, R14, #Carry_Flag`). Note that all the flags returned to the system are returned to the caller, so user's conditional code must be written with this in mind.

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

SWI decoding table

Pointer to table of SWI names

Offset in header

&24

Use

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

The table format is:

SWI group prefix
Name of 0th SWI
Name of 1st SWI
.
.
.
Name of nth SWI
0 byte to terminate

All names are null terminated. The group prefix is the first part of the full SWI name. ie. the first SWI's full name is <group prefix>_<name of 1st>. For example, the shell module's table is:

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

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

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

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

SWI decoding code

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

Offset in header

&28

On entry

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

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

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

On exit

R12 preserved

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

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

Interrupts

Interrupts are enabled on entry
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Entry point is not re-entrant

Use

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

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

OS_Byte 143 (SWI &06)

Issue module service call

On entry

R0 = 143 (&8F) (reason code)
R1 = service type
R2 = argument for service

On exit

R0 preserved
R1 preserved
R2 = may contain a return argument

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

Related SWIs

OS_ServiceCall (SWI &30)

Related vectors

ByteV

OS_Module (SWI &1E)

Perform a module operation

On entry

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

On exit

R0 preserved
other register states depends on the reason code

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

R0	Meaning
0	Run
1	Load
2	Enter
3	ReInit
4	Delete
5	Describe RMA
6	Claim
7	Free
8	Tidy
9	Clear
10	Insert module from memory
11	Insert module from memory and move into RMA
12	Extract module information
13	Extend block
14	Create new instantiation
15	Rename instantiation
16	Make preferred instantiation

17 Add expansion card module
18 Lookup module name
19 Enumerate ROM modules

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

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

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

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

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

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

Related SWIs

None

Related vectors	None

OS_Module 0 (SWI &1E)

Run

On entry

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

On exit

Does not return unless error occurs

Use

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

Possible errors are `File not found`, `No room in RMA`, `Not a module`, `Duplicate module refused to die`, and `Module refuses to initialise`.

Related reason codes

1, 2

OS_Module 1 (SWI &1E)

Load

On entry

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

On exit

R0 preserved
R1 preserved

Use

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

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

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

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

Possible errors are `File not found`, `No room in RMA`, `Not a module`, `Duplicate module refused to die`, and `Module refuses to initialise`.

Related reason codes

0, 2

OS_Module 2 (SWI &1E)

Enter

On entry

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

On exit

Does not return unless error occurs

Use

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

For a description of how a module is started up as an application, refer to OS_FSControl 2 (SWI &29).

Related reason codes

0

OS_Module 3 (SWI &1E)

Re-Initialise

On entry

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

On exit

R0 preserved
R1 preserved

Use

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

Related reason codes

8, 9

OS_Module 4 (SWI &1E)

Delete

On entry

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

On exit

R0 preserved
R1 preserved

Use

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

```
*RMKill FileCore%Base
```

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

Related reason codes

None

OS_Module 5 (SWI &1E)

Describe RMA

On entry

R0 = 5 (reason code)

On exit

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

Use

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

Related reason codes

6

OS_Module 6 (SWI &1E)

Claim

On entry

R0 = 6 (reason code)
R3 = required size

On exit

R0 preserved
R2 = pointer to claimed block
R3 preserved

Use

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

The possible error is `No room in RMA`.

Related reason codes

5, 7

OS_Module 7 (SWI &1E)

Free

On entry

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

On exit

R0 preserved
R2 preserved

Use

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

The possible error is `Not a heap block`.

Related reason codes

6

OS_Module 8 (SWI &1E)

Tidy

On entry

R0 = 8 (reason code)

On exit

R0 preserved

Use

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

Should any instantiation refuse to die (temporarily), and another module be called, then the module that has already been called with a non-fatal finalisation is re-initialised. If it cannot be re-initialised, then then that module is deleted from the system.

The SWI then exits with the original error. If it succeeds, then it collects the RMA together into one large unfragmented block and reinitialises the modules again. Any private words containing pointers to workspace blocks in the RMA are relocated. This should enlarge application space.

Related reason codes

3, 9

OS_Module 9 (SWI &1E)

Clear

On entry

R0 = 9 (reason code)

On exit

R0 preserved

Use

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

Related reason codes

3, 8

OS_Module 10 (SWI &1E)

Insert module from memory

On entry

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

On exit

R0 preserved
R1 preserved

Use

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

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

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

Related reason codes

11

OS_Module 11 (SWI &1E)

Insert module from memory and move into RMA

On entry

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

On exit

R0 preserved
R1 preserved
R2 preserved

Use

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

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

Related reason codes

10

OS_Module 12 (SWI &1E)

Extract module information

On entry

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

On exit

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

Use

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

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

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

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

Possible errors are `No more modules` or `No more instantiations`.

Related reason codes

13

OS_Module 13 (SWI &1E)

Extend block

On entry

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

On exit

R0 preserved
R2 = pointer to new allocated block
R3 preserved

Use

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

The possible error is `No room in RMA`.

Related reason codes

12

OS_Module 14 (SWI &1E)

Create new instantiation

On entry

R0 = 14 (reason code)
R1 = pointer to full name of new instantiation

On exit

R0 preserved
R1 preserved

Use

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

```
<Module name>%<instantiation name>
```

For example:

```
FileCore%RAM
```

Related reason codes

15, 16

OS_Module 15 (SWI &1E)

Rename instantiation

On entry

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

On exit

R0 preserved
R1 preserved
R2 preserved

Use

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

```
FileCore%RAM
```

to

```
FileCore%ADFS
```

Related reason codes

14, 16

OS_Module 16 (SWI &1E)

Make preferred instantiation

On entry

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

On exit

R0 preserved
R1 preserved

Use

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

Related reason codes

14, 15

OS_Module 17 (SWI &1E)

Add expansion card module

On entry

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

On exit

R0 preserved
R1 preserved
R2 preserved
R3 preserved

Use

This allows expansion card modules to be added to the module list.

Related reason codes

10

OS_Module 18 (SWI &1E)

Look-up module name

On entry

R0 = 18 (reason code)
R1 = pointer full module%instantiation name

On exit

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

Use

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

Related reason codes

12, 19

OS_Module 19 (SWI &1E)

Enumerate ROM modules

On entry

R0 = 19 (reason code)
R1 = module number (as returned by OS_Module 19)
R2 = -1 for ROM or expansion card number

On exit

R0 preserved
R1 = incremented
R2 preserved
R3 = pointer to module name
R4 = –1 (unplugged)
0 (inserted but not in the module chain ie dormant)
1 (active)
2 (running)
R5 = chunk number of expansion card module

Use

This returns information about the modules that are currently in ROM, along with their status.

Related reason codes

12, 18

OS_ServiceCall (SWI &30)

Issue a service call to a module

On entry

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

On exit

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

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

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

Number	Name	Meaning
&00	Service_Serviced	Service call has been claimed
&04	Service_UKCommand	Unknown command
&06	Service_Error	Error has occurred
&07	Service_UKByte	Unknown OS_Byte
&08	Service_UKWord	Unknown OS_Word
&09	Service_Help	*Help has been called
&0B	Service_ReleaseFIQ	Release FIQ
&0C	Service_ClaimFIQ	Claim FIQ
&11	Service_Memory	Memory controller about to be remapped
&12	Service_StartUpFS	Start-UpFiling System
&27	Service_Reset	Post-Reset
&28	Service_UKConfig	Unknown *Configure
&29	Service_UKStatus	Unknown *Status

&2A	Service_NewApplication	Application about to start
&40	Service_FSRedeclare	Filing system re-initialise
&41	Service_Print	For internal use only
&42	Service_LookupFileType	Lookup file type
&43	Service_International	International service
&44	Service_Keyhandler	Keyboard handler
&45	Service_PreReset	Pre-reset
&46	Service_ModeChange	Mode change
&47	Service_ClaimFIQinBackground	Claim FIQ in background
&48	Service_ReAllocatePorts	Econet restarting
&49	Service_StartWimp	Start the Wimp
&4A	Service_StartedWimp	Started the Wimp
&4B	Service_StartFiler	Start the Filer
&4C	Service_StartedFiler	Started the Filer
&4D	Service_PreModeChange	Mode change
&4E	Service_MemoryMoved	OS_ChangeDynamicArea has just finished
&4F	Service_FilerDying	Filer is dying
&50	Service_ModeExtension	Allow soft modes
&51	Service_ModeTranslation	Translate modes for unknown monitor types
&52	Service_MouseTrap	For non-click mouse warnings
&53	Service_WimpCloseDown	Trap WimpCloseDown calls
&54	Service_Sound	Parts of the Sound system starting/dying
&55	Service_NetFS	Either a *Logon or a *Bye has occurred
&56	Service_EconetDying	Econet about to leave
&57	Service_WimpReportError	Wimp is opening/closing a ReportError window

This section describes each of the above service calls or indicates where to find out information about the service calls that relate to a specific module.

Related SWIs

OS_Byte 143 (SWI &06)

Related vectors

None

Service_Serviced (Service Call &00)

Service call has been claimed

On entry

R1 = 0

On exit

R1 preserved

Use

This call is passed around following a successful claiming of a service call by a module.

Service_UKCommand (Service Call &04)

Unknown command

On entry

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

On exit

R0 = 0 for no error, else error pointer
R1 = 0 to claim the command, or preserved to pass on

Use

If you claim the call and execute the command successfully you should set R1 to 0. If an error occurs during execution then you should return with the pointer to the error buffer in R0. This call is issued after OSCLI has searched modules but before the filing system is called to try to *Run. It is also used to implement NetFS file server commands.

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

Service_Error (Service Call &06)

Error has occurred

On entry

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

On exit

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

Use

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

Service_UKByte (Service Call &07)

Unknown OS_Byte

On entry

R1 = &07 (reason code)
R2 = OS_Byte number
R3 = first parameter
R4 = second parameter

On exit

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

Use

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

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

Service_UKWord (Service Call &08)

Unknown OS_Word

On entry

R1 = & (reason code)
R2 = OS_Word number
R3 = pointer to OS_Word parameter block

On exit

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

Use

The same action applied as the OS_Byte entry.

Service_Help (Service Call &09)

*Help has been called

On entry

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

On exit

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

Use

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

Service_ReleaseFIQ (Service Call &0B)

Release FIQ

On entry

R1 = &0B (reason code)

On exit

R1 preserved

Use

This is issued immediately after the FIQ handler is released.

See the chapter entitled *Hardware vectors* for information about FIQ.

Service_ClaimFIQ (Service Call &0C)

Claim FIQ

On entry

R1 = &0C (reason code)

On exit

R1 preserved

Use

This is issued before the FIQ handler is claimed. It must only be issued from foreground tasks.

See the chapter entitled *Hardware vectors* for information about FIQ.

Service_Memory (Service Call &11)

For more information about this service call, refer to the chapter entitled *The Window Manager*

Service_StartUpFS (Service Call &12)

Start up Filing System

On entry

R1 = &12 (reason code)
R2 = filing system number

On exit

R1 preserved (never claim)
R2 preserved

Use

This is an old way to start up a filing system. It must not be claimed See the chapter on filing systems for the correct way to start up a filing system.

Service_Reset (Service Call &27)

Post-Reset

On entry

R1 = &27 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This is issued at the end of a machine reset. It must never be claimed.

Service_UKConfig (Service Call &28)

Unknown *Configure

On entry

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

On exit

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

Use

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

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

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

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

Note that it is also possible to trap unknown *Configure commands through the module's command table (see below). Only one of these mechanisms (and not this one by preference) should be used.

Service_UKStatus (Service Call &29)

Unknown *Status

On entry

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

On exit

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

Use

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

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

Note that it is also possible to trap unknown *Status commands through the module's command table – this is the preferred method.

Only one of these methods should be used.

Service_NewApplication (Service Call &2A)

Application about to start

On entry

R1 = &2A (reason code)

On exit

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

Use

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

Service_FSRedeclare (Service Call &40)

Filing system re-initialise

On entry

R1 = &40 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This service is called when the FileSwitch module has been re-initialised (due to an *RMReinit, for example). If you are in a filing system, you should make yourself known to FileSwitch by calling OS_FSControl 'add filing system' as described in the chapter on Filing systems. You must not claim this call.

Service_Print (Service Call &41)

This service call is for internal use only. You must not use it in your own code.

Service_LookupFileType (Service Call &42)

Lookup file type

On entry

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

On exit

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

Use

This call is passed round when FileSwitch would like to convert a twelve-bit file type into a textual name. If the file type passed in R2 is known to you, you should return with R1=0, and R2, R3 containing the eight characters in the name. If no-one claims the call, FileSwitch will convert the number into a three-digit hex value padded with spaces. This might be loaded as follows:

```
ADR    R1, nameString
       LDMIA R1, {R2,R3}
       MOV   R1, #0
       MOV   PC, R14
.nameString
       EQUS  "MY TYPE "
```

Service_International (Service Call &43)

International service

On entry

R1 = &43 (reason code)
R2 = sub-reason code
R3 - R5 depend on R2

On exit

R1 = 0 to claim, else preserved to pass on
R4 - 5 depend on R2 on entry

Use

This call should be supported by any modules which add to the set of international character sets and countries. It is used by the international system module * Command interface, and may be called by applications too. See the chapter entitled *International module* for details.

R2 contains a sub reason code which indicates which service is required:

R2	Service required
0	Convert country name to country number
1	Convert alphabet name to alphabet number
2	Convert country number to country name
3	Convert alphabet number to alphabet name
4	Convert country number to alphabet number
5	Define range of characters
6	Informative: New keyboard selected for use by keyboard handlers

Sub-reason code 0

The following pages give details of each of these sub-reason codes. Most users will not need to issue these service calls directly, but the OS_Byte calls and * Commands use these. The information is provided mainly for writers of modules whixh provide additional alphabets etc.

Service_International 0 (Service Call &43)

Convert country name to country number

On entry

R1 = &43 (reason code)
R2 = 0 (sub-reason code)
R3 = pointer to country name terminated by a null

On exit

R1 = 0 if claimed, otherwise preserved
R2 preserved
R3 preserved
R4 = country number recognised, preserved if not recognised

Use

Any module providing additional countries should compare the given country name with each country name provided by the module, ignoring case differences between letters and allowing for abbreviations using '.'. If the given country name matches a known country name, then it should claim the service (by setting R1 to 0), and set R4 to the corresponding country number.

If the given country name is not recognised, all registers should be preserved.

Service_International 1 (Service Call &43)

Convert alphabet name to alphabet number

On entry

R1 = &43 (reason code)
R2 = 1 (sub-reason code)
R3 = pointer to alphabet name terminated by a null

On exit

R1 = 0 if claimed, otherwise preserved
R2 preserved
R3 preserved
R4 = alphabet number recognised, preserved if not recognised

Use

Any module providing additional alphabets should compare the given alphabet name with each alphabet name provided by the module, ignoring case differences between letters and allowing for abbreviations using '.'. If the given alphabet name marches a known alphabet name, then it should claim the service (by setting R1 to 0), and set R4 to the corresponding alphabet number.

If the given alphabet name is not recognised, all registers should be preserved.

Service_International 2 (Service Call &43)

Convert country number to country name

On entry

R1 = &43 (reason code)
R2 = 2 (sub-reason code)
R3 = country number
R4 = pointer to buffer for name
R5 = length of buffer in bytes

On exit

R1 = 0 if claimed, otherwise preserved
R2 preserved
R3 preserved
R4 preserved
R5 = number of characters put into buffer if recognised, otherwise preserved

Use

Any module providing additional countries should compare the given country number with each country number provided by the module. If the given country number matches a known country number, then it should claim the service (by setting R1 to 0), put the country name in the buffer, and set R5 to the number of characters put in the buffer.

If the given country number is not recognised, all registers should be preserved.

Service_International 3 (Service Call &43)

Convert alphabet number to alphabet name

On entry

R1 = &43 (reason code)
R2 = 3 (sub-reason code)
R3 = alphabet number
R4 = pointer to buffer for name
R5 = length of buffer in bytes

On exit

R1 = 0 if claimed, otherwise preserved
R2 preserved
R3 preserved
R4 preserved
R5 = number of characters put into buffer if recognised, otherwise preserved

Use

Any module providing additional alphabets should compare the given alphabet number with each alphabet number provided by the module. If the given alphabet number matches a known alphabet number, then it should claim the service (by setting R1 to 0), put the alphabet name in the buffer, and set R5 to the number of characters put in the buffer.

If the given alphabet number is not recognised, all registers should be preserved.

Service_International 4 (Service Call &43)

Convert country number to alphabet number

On entry

R1 = &43 (reason code)
R2 = 4 (sub-reason code)
R3 = country number

On exit

R1 = 0 if claimed, otherwise preserved
R2 preserved
R3 preserved
R4 = alphabet number if recognised, otherwise preserved

Use

Any module providing additional countries should compare the given country number with each country number provided by the module. If the given country number matches a known country number, then it should claim the service (by setting R1 to 0), and set R4 to the corresponding alphabet number.

If the given country number is not recognised, all registers should be preserved.

Service_International 5 (Service Call &43)

Define a range of characters from a given alphabet number

On entry

R1 = &43 (reason code)
R2 = 5 (sub-reason code)
R3 = alphabet number
R4 = ASCII code of first character in range
R5 = ASCII code of last character in range

On exit

R1 = 0 if claimed, otherwise preserved
R2 preserved
R3 preserved
R4 preserved
R5 preserved

Use

Any module providing additional alphabets should compare the given alphabet number with each alphabet number provided by the module. If the given alphabet number matches a known alphabet number, then that service should be claimed (by setting R1 to 0) and all the characters should be defined in the range R4 to R5 inclusive, using calls to VDU 23. Any characters not defined in the specified alphabet are missed out: for example, characters &80–&9F in Latin1.

If the given alphabet number is not recognised, all registers should be preserved.

Service_International 6 (Service Call &43)

Notification of a new keyboard selection

On entry

R1 = &43 (reason code)
R2 = 6 (sub-reason code)
R3 = new keyboard number
R4 = alphabet number associated with this keyboard

On exit

R1 preserved (call must not be claimed)
R2 preserved
R3 preserved
R4 preserved

Use

This service call is for internal use by keyboard handlers. It is sent automatically after the keyboard selection is changed. You must not claim it.

Service_KeyHandler (Service Call &44)

Keyboard handler

On entry

R1 = &44 (reason code)
R2 = keyboard ID: 1 for A300 - A400 series keyboard

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.

Service_PreReset (Service Call &45)

Pre-Reset

On entry

R1 = &45 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is made just before a software generated reset takes place, when the user releases Break. This gives a chance for expansion card software to reset its devices, as this type of reset does not actually cause a hardware reset signal to appear on the expansion card bus. This call must not be claimed.

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_ClaimFIQinBackground (Service Call &47)

Claim FIQ in background

On entry

R1 = &47 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

RISC OS allows FIQ to be clamied in the background. Unlike foreground FIQ claim, background claim may fail. If you receive this call, you are the current FIQ owner, and you are not due to receive a FIQ, then the service should be claimed, after you have relinquished FIQ as usual.

Background claims are released by Service_ReleaseFIQ as before.

Service_ReAllocatePorts (Service Call &48)

Econet restarting

On entry

R1 = &48 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is made whenever Econet restarts. It is then up to the Econet software to allocate ports, set up TxCBs and RxCBs etc.

Service_StartWimp (Service Call &49)

For more information about this service call, refer to the chapter entitled *The Window Manager*

Service_StartedWimp (Service Call &4A)

For more information about this service call, refer to the chapter entitled *The Window Manager*

Service_StartFiler (Service Call &4B)

For more information about this service call, refer to the chapter entitled *The Window Manager*

Service_StartedFiler (Service Call &4C)

For more information about this service call, refer to the chapter entitled *The Window Manager*

Service_PreModeChange (Service Call &4D)

Mode change

On entry

R1 = &46 (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
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_MemoryMoved (Service Call &4E)

OS_ChangeDynamicArea has just finished.

On entry

R1 = &46 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is made whenever OS_ChangeDynamicArea (SWI &2A) has just finished. It is used by the Wimp to tidy up and should never be claimed.

Service_FilerDying (Service Call &4F)

For more information about this service call, refer to the chapter entitled *The Window Manager*

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.

Format of VIDC list (all word values):

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).

Format of workspace list (all word values):

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).

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 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_MouseTrap (Service Call &52)

For more information about this service call, refer to the chapter entitled *The Window Manager*

Service_WimpCloseDown (Service Call &53)

For more information about this service call, refer to the chapter entitled *The Window Manager*

Service_Sound (Service Call &54)

Parts of the the Sound system starting or dying

On entry

R0 = 0 – DMA handler starting
1 – DMA handler dying
2 – Channel handler starting
3 – Channel handler dying
4 – Scheduler starting
5 – Scheduler dying

R1 = &54 (reason code)

On exit

Registers preserved

Use

This call is made to signal that a part of the Sound system is about to start up or finish.

Service_NetFS (Service Call &55)

Either a *Logon or *Bye has occurred

On entry

R1 = &55 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is issued by NetFS to indicate to the NetFiler that things may have changed. For example, a user logged on to a server, while temporarily outside the Winp.

Service_EconetDying (Service Call &56)

Econet is about to leave

On entry

R1 = &56 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

This call is made whenever Econet is about to leave. It os then up to the Econet software to release ports, delete RxCBs and TxCBs etc.

Service_WimpReportError (Service Call &57)

For more information about this service call, refer to the chapter entitled *The Window Manager*

*Modules

Displays information about all installed relocatable modules

Syntax

```
*Modules
```

Parameters

—

Use

*Modules lists all the system modules in ROM and relocatable modules in RAM which are currently present in the machine.

The command displays the number allocated to each installed module (this may change as other modules are installed and removed), its position in memory, the address of its workspace, and its name.

System modules are stored in ROM, but may still be *UnPlugged, or replaced by RAM-based modules. The names listed by this command are the module titles which are supplied to other commands, eg *RMKill. This command also lists the base addresses and workspace areas of the modules.

Example

```
*Modules
No.    Position     Workspace    Name
  1    0380873C     00000000     UtilityModule
  2    0381FB94     01800014     FileSwitch
       .
       .
 38    01819034     00000000     SharedCLibrary
 39    0182CD74     01817304     ColourTrans
```

Related commands

*ROMModules

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMClear

Deletes relocatable modules from the relocatable module area of memory

Syntax

```
*RMClear
```

Parameters

—

Use

*RMClear deletes relocatable modules that you have specifically loaded into the RMA and frees their workspace. ROM resident modules are not deleted in this way. You must use *UnPlug for this.

RMCleared modules can be restored with *RMReInit.

Related commands

*RMReInit, *UnPlug, *RMTidy

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMEnsure

Checks the presence and version of a module

Syntax

```
*RMEnsure <moduletitle> <version number> [<*command>]
```

Parameters

`<moduletitle>`	the title of any currently installed module
`<version number>`	number against which the version number will be checked
`<*command>`	a Command Line command

Use

*RMEnsure checks that a module is present and is the given version (or a more recent one). A command, optionally given as a third parameter, is executed if the version is too old. This command is usually used in command scripts or programs to ensure that modules they need are loaded.

Example

```
*RMEnsure WindowManager 2.01 *rmload system:Wimp
```

Related commands

None

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMFaster

Makes a module faster by copying it from ROM to RAM

Syntax

```
*RMFaster <moduletitle>
```

Parameters

`<moduletitle>` the title of any ROM resident module

Use

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

Example

```
*RMFaster BASIC
```

Related commands

None

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMKill

Deactivates and deletes a relocatable module

Syntax

```
*RMKill <moduletitle>[<instantiation>]
```

Parameters

`<moduletitle>` the title of any currently installed module

Use

*RMKill deactivates the preferred instantiation of a relocatable module or the specified instantiation if the second argument is used, and releases its workspace. If located in RAM, it is also deleted. System modules are removed until the next hard reset or call to *RMReInit.

Example

```
*RMKill Debugger
```

Related commands

*RMLoad, *ROMModules, *UnPlug, *RMReInit

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMLoad

Loads and initialises a relocatable module

Syntax

```
*RMLoad <pathname> [<module init string>]
```

Parameters

`<pathname>`	the pathname to a file containing a valid module
`<module init string>`	optional parameters to the module

Use

*RMLoad loads and initialises a relocatable module. It is then available to the help system, and can provide * and SWI commands if available.

The optional initialisation can be used by certain modules to install themselves in a particular way. For example, it might give the amount of workspace that the module should claim, possibly overriding configuration information stored in CMOS RAM.

Note that a file loaded by this command (and *RMRun) must have file type FFA. If it doesn't, the module handler will refuse to load it.

Example

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

Related commands

*RMKill, *ROMModules, *UnPlug, *RMReInit, *RMRun

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMReInit

Reinitialises a relocatable module

Syntax

```
*RMREINIT <module title> [<module init string>]
```

Parameters

`<moduletitle>`	the title of any currently installed module, active or otherwise
`<module init string>`	optional parameters to the module

Use

*RMReInit reinitialises a relocatable module, reversing the action of *UnPlug, *RMKill or *RMClear on a ROM resident module. The module is returned to the state it was in when it was loaded.

Example

```
*RMReInit Debugger
```

Related commands

*UnPlug, *RMKill, *RMClear, *RMLoad, *ROMModules

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMRun

Runs a relocatable module

Syntax

```
*RMRun <pathname>
```

Parameters

`<pathname>`	valid pathname for a module file

Use

*RMRun loads and initialises a relocatable module. It is available to the help system, and can provide SWIs and * Commands. The module is then run, if it can be.

This call is equivalent to a call to *RMLoad followed by an enter operation in OS_Module. If the module is already resident, then it will simply be entered.

If a module cannot be run, then this command is equivalent to a *RMLoad command.

Example

```
*RMRun My_Module
```

Related commands

*RMKill, *ROMModules, *UnPlug, *RMReInit, *RMLoad

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*RMTidy

Compacts and reinitialises the RMA

Syntax

```
*RMTidy
```

Parameters

—

Use

*RMTidy compacts the RMA (Relocatable Module Area), reinitialising all the modules.

All free space is collected into a consecutive chunk of memory. This command must be used with extreme caution, as it is so drastic in its effects.

Related commands

*RMClear

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*ROMModules

Displays information about relocatable modules currently in ROM

Syntax

```
*ROMModules
```

Parameters

—

Use

*ROMModules lists all the system modules in ROM, whether part of the system or in expansion cards ('podules'), along with their status: active, dormant or unplugged.

System modules are stored in ROM, but may still be *UnPlugged, or replaced by RAM-based modules. The names listed by this command are the module titles which are supplied to other commands, eg *RMKill.

Example

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

Related commands

*Modules

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

*UnPlug

Disables a module

Syntax

`*Unplug [<moduletitle>]`

Parameters

`<moduletitle>` the title of any currently installed module

Use

The *Unplug command prevents the named ROM module from being initialised (and hence available for use). This setting is stored in the CMOS RAM, and is therefore permanent in its effects until a *RMReInit is issued. The workspace of the module is freed. You should use this command with caution, otherwise you may find programs stop working because you have unplugged a module that the program needs to use.

*Unplug without a module title displays a list of the unplugged ROM modules.

Example

`*Unplug RAMFSFiler` disables the RAMFSFiler module

Related commands

*RMReInit

Related SWIs

OS_Module (SWI &1E)

Related vectors

None

Program Environment

Introduction

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

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

Overview and Technical Details

Starting a task

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

- *RMRun the program
- OS_Module 'Enter' a module
- *Run the program
- *Go, to execute the program in memory

Modules

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

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

Programs on file

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

Programs in memory

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

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

Transient programs

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

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

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

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

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

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

Ending a task

Before describing the calls which control the application program's environment, it is worth explaining how to leave an application. In general, a simple 'return from subroutine' using MOV PC,R14 won't suffice. Instead, you should use a routine called OS_Exit (SWI &11). This passes control back to a well-defined place, which defaults to the supervisor * prompt, but could equally be a location in the previous application.

*Quit is equivalent to a call to OS_Exit.

OS_ExitAndDie (SWI &50) is like OS_Exit, but will kill a named module as well. This is used when a module is specific to a particular application.

System variables

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

Naming

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

Types

There are several types of system variable:

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

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

  ```
  *SetMacro CLI$prompt <sys$time><&20>
  ```

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

All the above types can be set with OS_SetVarVal (SWI &24) and read with OS_ReadVarVal (SWI &23).

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

Miscellaneous environment features

OS_GetEnv (SWI &10) is a multi-purpose SWI that provides three useful pieces of information:

1 address of the * Command string. This can be processed with OS_ReadArgs, which is described in the chapter entitled *Conversions*.

2 the real time that the program was started

3 the maximum amount of memory allocated for the program. This can be altered with reason code 0 of OS_ChangeEnvironment.

OS_WriteEnv (SWI &48) allows you to set the program start time and the command string.

Handlers

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

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

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

Before reading this section, you should be familiar with the chapters entitled *Software vectors* and *Hardware vectors*, since many of these handlers are directly called from these vectors

SWIs

OS_ChangeEnvironment (SWI &40) is the central SWI for handlers. There are several other routines that perform subsets of its actions. You are strongly recommended to use OS_ChangeEnvironment in any new applications as the others are only provided for compatibility.

The other calls are OS_Control (SWI &0F), OS_SetEnv (SWI &12), OS_CallBack (SWI &15), OS_BreakCtrl (SWI &18) and OS_UnusedSWI (SWI &19).

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

Details of Handlers

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

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

Undefined instruction, Prefetch abort, Data abort, Address exception

These handlers are all called from hardware vectors. See the chapter entitled *Hardware vectors* for a description of them. These handlers are all entered with the processor in SVC mode.

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

Error

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

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

The error buffer (the address of which should be set along with the handler address) contains the following:

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

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

BreakPoint

This handler is called when the SWI OS_BreakPt (SWI &17) is called. All the user mode registers are dumped into a buffer (the address of which should be set along with the handler address) and then the handler is entered in SVC mode. You can specify a pointer to workspace to pass in R12 when this handler is called.

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

```
ADR    R14, saveblock          get address of saved registers
LDMIA  R14, {R0  -  R14}^      load user registers from block
                               Note that user R13,R14 are altered
LDR    R14, [R14,#15*4];       load user PC into SVC R14
MOVS   PC, R14                 return to correct address and mode
```

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

Escape

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

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

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

To continue after an escape, the handler should reload the PC with the contents of R14. If R12 contains 1 on return then the CallBack handler will be used. Typically (eg for BASIC), the handler will set an internal flag which is checked by the foreground program.

Event

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

When the handler is entered the processor is in either SVC or IRQ mode, with the following register values:

R0	event reason code
R1...	parameters according to event code
R12	pointer to workspace, if set up – never contains 1
R13	a full, descending stack pointer

To continue after an event, the handler should reload the PC with the contents of R14. If R12 contains 1 on return then the CallBack handler will be used.

Exit

This handler is called when the SWIs OS_Exit (SWI &11) or OS_ExitAndDie (SWI &50) are called. It is entered with the processor in user mode. You can specify a pointer to workspace to pass in R12 when this handler is called.

Unused SWI

This handler is called by the default owner of the UKSWIV. (If RISC OS can't decode the number of a SWI into one which it supports directly, it offers it as a service call to modules. If none of them claim the service, it then calls the vector UKSWIV. This allows a user routine on that vector to try to deal with the SWI. If there is no such routine, or the one(s) that is present passes the call on, then the default owner of the vector calls the Unused SWI handler.)

You can specify a pointer to workspace to pass in R12 when this handler is called.

When the handler is entered the processor is in SVC mode, with interrupts in the same state as the caller. The registers have the following values:

R11	SWI number (Bit 17 clear)
R13	SVC stack pointer
R14	user PC with V cleared

R10, R11 and R12 are stacked and are free for your own use.

UpCall

This handler is called by the default owner of UpCallV when OS_UpCall (SWI &33) is called. OS_UpCall (SWI &33) is used to warn your program of errors and situations that you may be able to recover from. See the chapters entitled *Software vectors* and *Communications within RISC OS* in the introductory part of this manual. You can specify a pointer to workspace to pass in R12 when this handler is called.

CallBack

This handler is called whenever RISC OS's internal CallBack flag is set, and the system next exits to User mode with interrupts enabled. It uses a buffer (the address of which should be set along with the handler address) in which all the registers are dumped when the handler is called. You can specify a pointer to workspace to pass in R12 when this handler is called. A more detailed description follows.

Callbacks in more detail

There are two types of CallBack usage under RISC OS:

- Transient callbacks are placed in a list by calling OS_AddCallBack (SWI &54). They are used to deal with a specific case, and are called once before being removed.
- The callback handler is permanent and takes all callbacks that are not intercepted by transients. These CallBacks are explicitly requested by calling OS_SetCallBack (SWI &1B). They can also be implicitly requested by setting R12 to 1 on exit from either an escape or event handler. There is a system default CallBack handler, but you can of course replace it using OS_ChangeEnvironment.

Transient CallBacks

Transient callbacks may be called on the system being threaded out of – that is, when it enters User mode with interrupts enabled. They can also be called when RISC OS is idling; for example, while it is waiting in OS_ReadC.

Transient CallBacks are usually set up by an interrupt routine that needs to do complex processing that would take too long in an interrupt, or that needs to call a non-re-entrant SWI. OS_AddCallBack tells RISC OS that the interrupt routine wishes to be 'called back' when the machine is in a state that no longer imposes the restrictions associated with an interrupt routine.

Transient CallBacks can safely be used by many clients.

Other CallBacks

The CallBack handler is only ever called on the system being threaded out of – that is, when it enters User mode with interrupts enabled. Unlike transient CallBacks, it is not called when RISC OS is idle. This means that you cannot rely on being called back within any given time. You must take this into consideration before using a CallBack handler.

Also, you must not allow a second CallBack before your first one has completed; see the *Application Notes* at the end of this chapter for an example of how to implement a semaphore to prevent this.

The CallBack code is called in IRQ or supervisor mode with interrupts disabled. The PC stored in the save block will be a user mode PC with interrupts enabled. Note that if the currently active program has interrupts disabled or is running in supervisor mode, CallBack is not used.

In the simple case the CallBack routine should be exited by:

```
ADR    R14, saveblock          get address of saved registers
LDMIA  R14, {R0  -  R14}^      load user registers from block
                               Note that user R13,R14 are altered
LDR    R14, [R14,#15*4];       load user PC into SVC R14
MOVS   PC, R14                 return to correct address and mode
```

Currently active object pointer

This is a pointer to the address of: the last application started the last error handler called the last exit handler called It is used by OS_Module to determine whether a module can be killed.

Setting up and restoring the environment

In order to deal correctly with the various ways in which applications can be run, and killed off, the following approach has been developed for setting up the program environment when an application starts, and restoring it when it is killed.

The basic problem is that if a new application is started 'on top' of the currently active one, it should completely replace the first, and should therefore have the same 'parent' environment as the first application. In order for this to happen, run-time language libraries and BASIC must be written so that they get themselves out of the way as a new application is started up in the same task space.

This also applies to machine-code programs which run as applications, for example modules which run as wimp tasks.

There are two possible approaches:

- Do not set up **any** handlers **at all**, and **always** call the 'X' form of SWIs, to avoid calling the error handler. If the error handler is called, the application will be terminated, as the parent error handler will be invoked.

- Set up Error, Exit and UpCall handlers as described below, so that the program environment can be restored correctly when the program terminates. You **must** provide all three of these handlers if you use any handlers at all.

Starting an application

When you start an application, you must:

4 Check that there is sufficient memory to start up – if not, call OS_GenerateError ("Not enough application memory")

5 Set up your handlers using the SWI XOS_ChangeEnvironment; store the values returned in R1-R3 so you can later restore the old handlers.

Note that you must store the previous values not only for Exit, Error and UpCall handlers, but also for any other handlers that are set up.

If your Error handler is called

If your error handler is called and you want to call the 'external' error handler (eg. BASIC if '–quit' was on the command line), you should:

6 restore all handlers to their original values (R1 - R3 for each)

7 call OS_GenerateError

If your Exit handler is called

If your exit handler is called you should:

8 restore all handlers to their original values (R1 - R3 for each)

9 call OS_Exit

If your UpCall handler is called

If your UpCall handler is called and R0 = UpCall_NewApplication (256), you should:

10 restore all handlers to their original values (R1 - R3 for each)

11 return to the caller, preserving all registers (ie carry on and start the new application)

The approach described ensures that it is not possible for the application to be terminated without it first restoring the handlers to their original values.

OS_Control (SWI &0F)

Read/write handler addresses

On entry

R0 = address of error handler, or 0 to read
R1 = pointer to buffer for the error handler, or 0 to read
R2 = address of escape state change handler, or 0 to read
R3 = address of event handler, or 0 to read

On exit

R0 = previous error handler address
R1 = previous buffer address
R2 = previous escape routine address
R3 = previous event handler address

Interrupts

Interrupts are not enabled
Fast interrupts are enabled

Processor Mode

Processor is in IRQ or SVC mode

Re-entrancy

SWI cannot be re-entered as interrupts are disabled

Use

OS_Control sets some of the exception handlers. The addresses of the error handler, error handler buffer, escape state change handler and event handler are passed in R0 - R3. Zero for any of these means no change. ie. a read-only operation.

Note that the call OS_ChangeEnvironment provides more control over the handlers than this call, and should be used in preference. (OS_Control actually uses OS_ChangeEnvironment.)

Related SWIs

OS_ChangeEnvironment (SWI &40)

Related vectors

ChangeEnvironmentV

OS_GetEnv (SWI &10)

Read environment parameters

On entry

—

On exit

R0 = address of the * Command string
R1 = permitted RAM limit (ie. highest address available + 1)
R2 = address of the real time the program was started (5 bytes)

Interrupts

Interrupt status is unaltered
Fast interrupt status is unaltered

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI reads some information about the program environment. The value returned in R0 is the address of a copy of the command line. R1 returns the address of the byte above the last one available to the application. The five bytes pointed to by R2 give the real time. ie. centiseconds since 00:00:00 01–Jan–1900.

The memory limit described in R1 can be altered by reason code 0 of OS_ChangeEnvironment.

OS_WriteEnv allows you to set these values.

Related SWIs

OS_WriteEnv (SWI &48), OS_ChangeEnvironment (SWI &40)

Related vectors

None

OS_Exit (SWI &11)

Pass control to the most recent exit handler

On entry

R0 = pointer to error block
R1 = "ABEX" (&58454241) if return code is to be set
R2 = return code

On exit

never returns

Interrupts

Interrupt status is unaltered
Fast interrupt status is unaltered

Processor Mode

Processor is in USR mode

Re-entrancy

SWI is not re-entrant

Use

When OS_Exit is called, control returns to the most recent exit handler address. The BASIC statement QUIT performs an OS_Exit. Before executing OS_Exit, however, you should restore any of the handlers changed in starting the application.

If the exiting program wishes to return with a result code, it must set R1 to the hex value shown above, and R2 to the desired value. Non-error results must be in the range 0 to the value of the variable Sys$RCLimit. The return value is assigned to the variable Sys$ReturnCode, which can be interrogated by any program using OS_ReadVarVal.

To return with an error, exit with a value less than zero or greater than Sys$RCLimit (having restored the previous error handler, as indicated above). This gives the error `Return code limit exceeded` (&1E2), but still sets the variable to the required value.

Related SWIs

OS_ExitAndDie (SWI &50)

Related vectors

None

OS_SetEnv (SWI &12)

Set environment parameters

On entry

R0 = address of the handler for OS_Exit, or 0 to read
R1 = address of the end of memory limit for OS_GetEnv to read, or 0 to read
R4 = address of handler for undefined instructions, or 0 to read
R5 = address of handler for prefetch abort, or 0 to read
R6 = address of handler for data abort, or 0 to read
R7 = address of handler for address exception, or 0 to read

On exit

R0 = address of previous handler for OS_Exit
R1 = address of previous end of memory limit for OS_GetEnv to read
R4 = address of previous handler for undefined instructions
R5 = address of previous handler for prefetch abort
R6 = address of previous handler for data abort
R7 = address of previous handler for address exception

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_SetEnv sets several of the handlers for a program.

Note that the call OS_ChangeEnvironment provides a superset of the facilities that this call provides, and should be used in preference. In fact, this call uses OS_ChangeEnvironment.

Related SWIs

OS_ChangeEnvironment (SWI &40)

Related vectors

None

OS_CallBack (SWI &15)

Set-up the CallBack handler

On entry

R0 = address of the register save block, or 0 to read
R1 = address of the CallBack handler, or 0 to read

On exit

R0 = address of previous register save block
R1 = address of previous CallBack handler

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_CallBack sets up the address of the CallBack handler and the register save block, zero for either value meaning no change.

Note that the call OS_ChangeEnvironment provides more control over the handlers than this call, and should be used in preference. (OS_CallBack actually uses OS_ChangeEnvironment.)

Related SWIs

OS_ChangeEnvironment (SWI &40)

Related vectors

ChangeEnvironmentV

OS_BreakPt (SWI &17)

Cause a break point trap to occur and the BreakPoint handler to be entered

On entry

—

On exit

—

Interrupts

Interruptstatus is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

When OS_BreakPt is executed, all the user mode registers are saved in a block and the BreakPoint handler is called. The saved registers are only guaranteed to be correct for user mode.

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

This SWI would be placed in code by the debugger at required breakpoints.

Related SWIs

OS_BreakCtrl (SWI &18)

Related vectors

None

OS_BreakCtrl (SWI &18)

Setup the BreakPoint handler

On entry

R0 = address of the register save block, or 0 for no change
R1 = address of the control routine, or 0 for no change

On exit

R0 = address of previous register save block
R1 = address of previous control routine
V is always clear

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_BreakCtrl sets up the address of the BreakPoint handler and the register save block, zero for either value meaning no change.

Note that the call OS_ChangeEnvironment provides more control over the handlers than this call, and should be used in preference. (OS_BreakCtrl actually uses OS_ChangeEnvironment.)

Related SWIs

OS_BreakPt (SWI &17)

Related vectors

ChangeEnvironmentV

OS_UnusedSWI (SWI &19)

Set-up the handler for unused SWIs

On entry

R0 = address of the unused SWI handler; or 0 for no change

On exit

R0 = address of previous unused SWI handler

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not enabled

Use

OS_UnusedSWI sets up the address of the UnusedSWI handler, zero meaning no change.

Note that the call OS_ChangeEnvironment provides more control over the handlers than this call, and should be used in preference. (OS_UnusedSWI actually uses OS_ChangeEnvironment.)

Related SWIs

OS_ChangeEnvironment (SWI &40)

Related vectors

ChangeEnvironmentV

OS_SetCallBack (SWI &1B)

Cause a call to the CallBack handler

On entry	—
On exit	—
Interrupts	Interrupts are disabled Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI cannot be re-entered because interrupts are disabled
Use	OS_SetCallBack sets the CallBack flag and so causes entry to the CallBack handler when the system next exits to user mode code with interrupts enabled (apart, of course, from the exit from this SWI). This SWI may be used if the code linked into the system (via a vector or as a SWI handler, etc) is required to do things on exit from the system.
Related SWIs	OS_CallBack (SWI &15)
Related vectors	None

OS_ReadVarVal (SWI &23)

Read a variable value

On entry

R0 = pointer to name, may be wildcarded (* and #)
R1 = pointer to buffer
R2 = maximum length of buffer
R3 = name pointer (or 0 for first call).
R4 = 3 if an expanded string is to be returned

On exit

R0, R1 preserved
R2 = number of bytes read
R3 = new name pointer, string is null-terminated
R4 = type of variable (string, number or macro)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ReadVarVal reads a variable and returns its value and its type. On entry, R3 should be 0 the first time the call is made for a wildcarded name, and thereafter preserved from the previous call. This enables all matches of a wildcarded name to be found. On exit, R3 points to the name of the variable found. The XOS_ReadVarVal form of the call should be used if you don't want an error to occur after the last name has been found.

You can call XOS_ReadVarVal to check for the existence of a variable by setting R2 to a value less than zero (bit 31 set) on entry. If it is still negative on exit, the variable exists; if it is zero, the variable does not exist.

The type of the variable read is returned in R4 as follows:

Value		Type
VarType_String	(0)	String
VarType_Number	(1)	4 byte (signed) integer
VarType_Macro	(2)	Macro

R4, if set to 3 on entry, indicates that a suitable conversion to a string should be performed. String variables are unaltered, numbers are converted to (signed) decimal strings, and macros are OS_GSTransed.

If R4 isn't 3 on entry, the un-OS_GSTransed version of a macro is returned, and the four-byte binary of a number is returned.

See the application notes at the end of this chapter for an example of reading a variable.

Related SWIs

OS_SetVarVal (SWI &24)

Related vectors

None

OS_SetVarVal (SWI &24)

Write a variable value

On entry

R0 = pointer to name. This can be wildcarded for update/delete
R1 = pointer to value
R2 = length of value. Negative means destroy the variable
R3 = name pointer or 0 for first call
R4 = type

On exit

R0 = preserved
R1 = preserved
R2 = preserved
R3 = new context pointer
R4 = type created if expression is evaluated

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

OS_SetVarVal either creates, updates or destroys a variable. The name may be terminated by any character whose ASCII value is 32 or less and may be wildcarded if it is to be updated or deleted (ie. if it already exists).

The pointer to the value to be assigned in the case of create/update is given by R1. If it is a string then it must be terminated by a linefeed (ASCII 10) or carriage return (ASCII 13). The interpretation of the value depends on the type given in R4 as follows:

Value	Type	
VarType_String	(0)	OS_GSTrans the given value
VarType_Number	(1)	Value is a 4 byte (signed) integer
VarType_Macro	(2)	Copy value (may be OS_GSTransed on use)

VarType_Expanded	(3)	The value is a string which should be evaluated as an expression using OS_EvaluateExpression, and assigned to a number or string variable, depending on the expression type
VarType_Code	(16)	Special case (see below)

If the call is successful, R3 is updated to point to the new context so allowing the next match of a wildcarded name to be obtained on a subsequent call. R4 returns the type created if an expression was evaluated (ie. if R4 was 3 on entry).

R2 must be negative on entry to delete a variable. Also, to delete a type-16 variable, R4 should contain 16 on entry.

VarType_Code

When R4 is set to 16 on entry (and R2 >= 0) a code variable may be created. In this case R1 is the pointer to the code fragment associated with the variable, and R2 is the length of the code fragment. This code must be word-aligned and takes the following format:

Offset	Contents
0	Branch instruction to entry point for write operation
4	Entry point for read operation
8...	Body of code
	...

Values are always written to (and read from) code variables as strings. The entry for the write operation is called whenever the variable is to be set, as follows:

On entry

R1 = pointer to the value to be used
R2 = length of value

On exit

R1, R2, R4, R10 - R12 may be corrupted

The entry for the read operation is called whenever the variable is to be read by a call to OS_ReadVarVal, as follows:

On entry

—

On exit

R0 = pointer to value
R1 = corrupted
R2 = length of value

Both entries are called in SVC mode. Therefore if any SWIs are used, R14 must be saved on the stack so that it does not become corrupted.

See the application notes at the end of this chapter for an example of a code variable.

Note that when a function key is input, the appropriate variable key$n is read using OS_ReadVarVal. Therefore by creating your own code variables with these names, you can cause the reading of a function key to cause a routine to be called instead of just a string being read.

Errors

OS_SetVarVal can return the following errors:

• Bad name	Wildcards/control characters in name when creating
• Bad string	OS_GSTrans unable to translate string
• Bad macro value	Control characters in the value string (R1)
• Bad expression	Expression cannot be evaluated
• Variable not found	For deletion or update
• No room for variable	Not enough room to create/update it (system heap full)
• Variable value too long	Variables are limited to 256 bytes
• Bad variable type	

Related SWIs

OS_ReadVarVal (SWI &23)

Related vectors

None

OS_ChangeEnvironment (SWI &40)

Install a handler

On entry

R0 = handler number
R1 = new address, or 0 for no change
R2 = R12 with which to call the routine, or 0 for no change
R3 = buffer pointer, if appropriate

On exit

R0 preserved
R1 = previous address
R2 = previous R12
R3 = previous buffer pointer

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

OS_ChangeEnvironment is a single routine which performs the actions of OS_Control, OS_SetEnv, OS_CallBack, OS_BreakCtrl, and OS_UnusedSWI. In fact, all of those routines use this call. In new programs, you should always use this call in preference to the earlier ones.

For full details of the handlers, see the section earlier in this chapter.

On entry, R0 contains a code which determines which particular handler's address is to be set up. The new address is passed in R1. R0 also determines whether R2 and R3 are relevant or not. This is summarised in the table below:

R0	Handler	R2	R3
0	MemoryLimit	Ignored	Ignored
1	Undefined ins.	Ignored	Ignored
2	Prefetch abort	Ignored	Ignored
3	Data abort	Ignored	Ignored
4	Address exception	Ignored	Ignored
5	Other exceptions	Ignored	Ignored
6	Error	R0 when called	Error buffer address
7	CallBack	R12 when called	Register buffer address
8	BreakPoint	R12 when called	Register buffer address
9	Escape	R12 when called	Ignored
10	Event	R12 when called	Ignored
11	Exit	R12 when called	Ignored
12	Unused SWI	R12 when called	Ignored
13	Exception registers	Ignored	Ignored
14	Application space	Ignored	Ignored
15	Currently active object	Ignored	Ignored
16	UpCall	R12 when called	Ignored

'Other exceptions' (handler 5) is for future expansion.

Handler 13 sets the address of the area in memory where the registers are dumped when one of the exceptions (1 - 5) occurs, if the default handlers are used.

Note that in order to perform its function, OS_ChangeEnvironment vectors through ChangeEnvironmentV. A routine linked onto this vector can stop the change from happening by setting R1 (and if appropriate R2, R3) to zero and passing the call on. See also the chapter entitled *Software vectors*.

Related SWIs

OS_Control (SWI &0F), OS_SetEnv (SWI &12), OS_CallBack (SWI &15) OS_BreakCtrl (SWI &18), OS_UnusedSWI (SWI &19)

Related vectors

ChangeEnvironmentV

OS_WriteEnv (SWI &48)

Set the program environment command string and start time

On entry

R0 = pointer to environment string
R1 = pointer to start time

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call sets the string that an application would read as its command string, containing parameters for the application. This SWI also sets the start time, which is the real-time, stored as a 5 byte value.

This SWI is mainly used for debuggers.

Related SWIs

OS_GetEnv (SWI &10)

Related vectors

None

OS_ExitAndDie (SWI &50)

Pass control to the most recent exit handler and kill a module

On entry

R0 = pointer to error block
R1 = "ABEX" (&58454241) if return code is to be set
R2 = return code
R3 = pointer to module name

On exit

never returns

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI is like OS_Exit, except that it will kill a module before exiting. R3 points to a string containing its name.

Related SWIs

OS_Exit (SWI &11)

Related vectors

None

OS_AddCallBack (SWI &54)

Add a transient callback to the list

On entry

R0 = address to call
R1 = value of R12 to be called with

On exit

R0 = preserved
R1 = preserved

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

A transient callback is placed on a list of tasks who want to be called as soon as RISC OS is not busy. Usually, this will be just before returning from a SWI or while waiting for a key and so on.

This SWI will place a transient routine on that list. It is usually called from an interrupt routine that needs to do complex processing that would take too long in an interrupt, or that needs to call a non-re-entrant SWI. It is usually called from an interrupt routine that needs to do complex processing that would take too long in an interrupt, or that needs to call a non-re-entrant SWI. Note that it is not necessary to call OS_SetCallBack. Using this SWI means you want to be called. OS_SetCallBack is only needed when using the callback handler.

A routine called by this mechanism must preserve all registers and return by MOV PC, R14.

Related SWIs

None

Related vectors

None

OS_ReadDefaultHandler (SWI &55)

Get the address of the default handler

On entry

R0 = reason code (0 - 16)

On exit

R0 = preserved
R1 = address of default handler
R2 = workspace address
R3 = buffer address

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

Using the same handler number in R0 as those in OS_ChangeEnvironment, this SWI returns details about the default handler.

Zero in R1, R2 or R3 on exit means that it is not relevant.

Related SWIs

OS_ChangeEnvironment (SWI &40)

Related vectors

None

*Go

Calls machine code at a given address

Syntax

```
*Go [<hexadecimal address>] [ ; environment]
```

Parameters

`<hexadecimal address>`	address of machine code to call
`; environment`	string to pass to machine code

Use

This command is followed by the address of the machine code to call. If the address is omitted, it defaults to &8000, which is where application programs (such as the C compiler) are loaded.

*Go enters an application, and you cannot use it to run machine code subroutines.

Example

```
*Go 9000 ; SrcList
```

Call machine code at &9000, passing it the string 'SrcList'

Related commands

None

Related SWIs

None

Related vectors

None

*Quit

Exit from current application

Syntax `*Quit`

Parameters None

Use Exits from the current application – that is, it returns to the previous context.

Related commands *GOS

Related SWIs None

Related vectors None

*Run

Loads and executes a file

Syntax

```
*Run <filename> [<parameters>]
```

Parameters

<filename>	a valid pathname specifying a file
<parameters>	a Command Line tail

Use

*Run loads and executes the named file, together with a list of parameters, if appropriate.

The filename which is supplied with the *Run command is searched for in the directories listed in the system variable Run$Path. By default, Run$Path is set to ',%.'. This means that the current directory is searched first, followed by the library.

The file's type (BASIC, Text etc) is looked for:

- If the file has no file type, it is loaded at its own load address, and execution commences at its execution address.
- If the file has a file type of &FFC or &FF8 it is treated specially – see the chapter entitled *FileSwitch*.
- Otherwise the Alias$@RunType variable corresponding to the file type is looked up to determine how the file is to be run. A BASIC file has a file type of &FFB, so the variable Alias$@RunType_FFB is looked up, and so on.

Example

```
*Run my_prog
```

```
*Run my_prog my_data
```

my_data is passed as a parameter to the program my_prog. The program can then use this filename to look up the data it needs.

Related commands

*Load, *Exec

Related SWIs

None

Related vectors

None

*Set

Assigns a string value to a system variable

Syntax

```
*Set <varname> <value>
```

Parameters

<varname> a variable name, or a wildcard specification for a single variable name

<value> this parameter depends on the system variable referred to in the <varname> specification, and is *GSTransed* before use

Use

*Set assigns a string value to a system variable, like an assignment statement in a programming language. For example:

```
*Set varname text
```

assigns the string 'text' to the variable varname.

Another use for the *Set command is to change the name of a command to one which is more convenient for the user:

```
*Set Alias$<name> <cname>
```

establishes <name> as an alternative name for the command <cname>; for example after:

```
*Set Alias$Aid Help
```

the command *Aid is now a synonym for *Help; both commands access the help system.

The command *Show Alias$* lists all aliases. Another example is:

```
*Set Alias$Mode Echo |<22>|<%0>
*Mode 12
```

The command implements a new command Mode, which sets the screen to mode 12 (in the above case). The Echo command reflects the string which follows it; |<22> generates the ASCII character 22, Ctrl V, which is equivalent to the VDU command to change mode. |<%0> reads the first parameter from the command line, and generates the corresponding ASCII code.

Example	`*Set Sys$Year 1988`
Related commands	*SetEval, *SetMacro, *Unset
Related SWIs	OS_GSTrans (SWI &27)
Related vectors	None

*SetEval

Evaluates an expression, assigning it to a system variable

Syntax

```
*SetEval <varname> <expression>
```

Parameters

`<varname>`	a valid variable name
`<expression>`	a valid Command Line expression

Use

*SetEval evaluates an expression and assigns the value to a system variable.

Example

```
*Set rate 12
*SetEval rate rate + 1
*Show rate
rate (Number) : 13

*SetEval fred "jim"+"sheila"
```

Related commands

*Set, *SetMacro, *Unset, *Eval

Related SWIs

None

Related vectors

None

*SetMacro

Assigns a macro value

Syntax

```
*SetMacro <varname> <value>
```

Parameters

`<varname>`	a valid variable name
`<value>`	value to be GSTransed at read time

Use

*SetMacro assigns a value to a system variable. The parameters making up the value are not interpreted when the command is given, but each time the variable is used.

Example

```
*SetMacro CLI$Prompt "<Sys$Time> "
13:43:17          system time replaces existing prompt
13:43:19          Return pressed two seconds later
```

This resets the Command Line prompt, which appears as the first item on each line, to be the current time whenever the prompt is given. Compare this with using the `*Set` command:

```
*Set fred <Sys$Time>
*Show fred
FRED  :  13:43:59
```

the `*Show` command issued five minutes later will produce:

```
*Show fred
FRED  :  13:43:59
```

Notice that the time is fixed at the time the `*Set` command is last used, in contrast to the `*SetMacro` command.

Related commands

*Set, *SetEval, *Unset

Related SWIs

None

Related vectors

None

*Show

Displays the list of system variables

Syntax

```
*Show [<variablespec>]
```

Parameters

`<variablespec>`	a variable name or a wildcard specification for a set of variable names

Use

`*Show <name>` displays the name, type and current value of any system variables matching the name given as a parameter. These include the 'special' system variables, which may be altered, but which cannot be deleted.

If no name is given, all system variables are displayed.

Example

`*Show`	lists all system variables
`*Show CLI$Prompt`	
`*Show Alias$*`	lists all aliases

Related commands

*Set, *SetEval, *SetMacro

Related SWIs

None

Related vectors

None

*Unset

Deletes a system variable

Syntax

```
*Unset <varname>
```

Parameters

`<varname>`	any currently Set system variable, which may be specified using wildcards.

Example

```
*Unset My_var
```

Related commands

*Set, *SetEval, *SetMacro

Related SWIs

None

Related vectors

None

Application Notes

Reading a variable

Here is a short example of reading a variable using OS_ReadVarVal:

```
;Print all sys$ variable names
        ADR     R1, valBuffer           ;Buffer to place value
        MOV     R3, #0                  ;Initial context
.loop
        ADR     R0, strName             ;Wildcarded name to find
        MOV     R2, #bufferLen          ;Length of value buffer
        SWI     "XOS_ReadVarVal"        ;Non-error reporting one
        MOVVSS  PC,R14                  ;Return and clear V
        MOV     R0, R3                  ;Get address of name
        SWI     "OS_Write0"             ;Print it
        SWI     "OS_NewLine"            ;and new line
        B       loop                    ;again
        ....
.strName        EQUS    "SYS$*" + CHR$0
```

Code variable

Below is a complete example of a program to create a variable called Mode. The read action is to return the current display mode, and the write action to to set the mode.

```
.start  ADR     R0, varName             ;Pointer to the name
        ADR     R1, code                ;Start of code body
        MOV     R2, #endCode-code ;Length of code body
        MOV     R3, #0                  ;Context pointer
        MOV     R4, #&10                ;'special' type
        SWI     "OS_SetVarVal"          ;Create it
        MOV     PC, R14                 ;Return

.code
        B       writeCode               ;Branch to write code
.readCode
        STMFD   R13!, {R14}             ;Save return address
        MOV     R0, #&87                ;OS_Byte read mode number
        SWI     "XOS_Byte"
        MOV     R0, R2                  ;Mode in R0 for conversion
        ADR     R1, buffer              ;Buffer for ASCII conversion
        MOV     R2, #4                  ;Max len of buffer
        SWI     "XOS_BinaryToDecimal"
        MOV     R0, R1                  ;Pointer in R0
                                        ;length already in R2
        LDMFD   R13!, {PC}              ;Return

.writeCode
        STMFD   R13!, {R14}             ;Save return address
        SWI     "XOS_ReadUnsigned"      ;R1 set correctly already
```

```
        SWIVC   &20100+22               ;VDU mode change
        MOVVC   R0,R2                   ;Get integer read in R0
        SWIVC   "XOS_WriteC"            ;Do mode change
        LDMFD   R13!, {PC}              ;Return

.buffer
        EQUD    0                       ;Buffer for string conversion
.endCode

.varName
        EQUS    "Mode "                 ;Name of variable
```

The routine at 'start' creates the variable. Obviously as the code body is copied into the system heap, it must be position independent. The two routines `readCode` and `writeCode` are called whenever an access to the variable is made. For example, a *Set Mode command will call the write code entry, and *Show sys$mode or *Echo <Mode> will call the read entry.

Notice that in the body of the code variable, only XOS_ SWIs are used. This is because it is important that errors are not generated when the read or write code executes. A more rigorous version of the code above would check V after each SWI and return if it was set.

OS_AddCallBack

The next example shows the use of OS_AddCallBack; it prints "Run away!" after 2 seconds:

```
DIM code 100
P%=code
[
.alarm STMFD r13!, {r14}
       SWI "XOS_WriteS"
       EQUS "Run away!"
       EQUB 10:EQUB 13:EQUB 0
       ALIGN
       LDMFD r13!, {r15}
.timer STMFD r13!, {r0,r14}
       MOV r0, r12           ; set up for us by BASIC bit
                             ; r12 is not used in alarm, so r1 here is don't-care
       SWI "XOS_AddCallBack"
       LDMFD r13!, {r0, r15}
]
SYS "OS_CallAfter",200,timer,alarm
```

A CallBack handler

The final example shows a callback handler, with a semaphore to prevent recursive callback; it prints "Run away!" when mouse buttons are pressed.

```
DIM code 200
P%=code
[
.sema      EQUB 1
           ALIGN
.saveblock :]:P%=P%+16*4:[
.callback ; entered here in a privileged mode, with interrupts disabled
           ; first thing to do is enable IRQs
           TEQP r15, #3   ; force SVC mode, IRQs on.
           SWI "XOS_WriteS"
           EQUS   "Run away!"
           EQUB 10:EQUB 13:EQUB 0
           ALIGN
           ADR r14, saveblock
           LDMIA r14, {r0-r14}^    ; most registers reloaded
           TEQP r15, #3+(1<<27)    ; disable IRQs for sema update
           MOVNV r0, r0            ; and return
           MOV r14, #1
           STRB r14, sema
           LDR  r14, saveblock+15*4  ; must not allow another callback request
                                     ; until the stashed PC is safe
           MOVS r15, r14             ; return, enableing IRQs etc
.events    CMP r0, #10               ; mouse button state change?
           MOVNES r15, r14           ; no - run away
           STMFD r13!, {r14}
           LDRB r12, sema            ; possibly request callback
           MOV r14, #0
           STRB r14, sema            ; and disable any futher requests
           LDMFD r13!, {r15}         ; until that one serviced.
]
SYS"OS_CallBack",saveblock,callback,0 TO osave,ocall
REM note that we aren't using r12 in the callback handler;
REM if this was in a module, for example, sema would be in the workspace,
REM and we would have to access it r12-relative; r12 would therefore be
REM set to be the workspace pointer on entry.
SYS "OS_ChangeEnvironment",10,events,0 TO ,oldev
*fx 14,10
REPEAT UNTILINKEY -1: REM loop until shift
*fx 13,10
SYS "OS_ChangeEnvironment",10,oldev,0
SYS "OS_CallBack",osave,ocall,0
REM note that in both the above calls, the R12 values are explicitly left
REM alone, because we didn't use them earlier.
```

Memory Management

Introduction

This chapter describes the memory management in RISC OS. This covers memory allocation by a program or module as well as using the MEMC chip to handle how memory is mapped.

In many environments, such as BASIC and C, you can use the language's intrinsic memory allocation routines, which use the calls described in this chapter transparently. For information, refer to Wimp_SlotSize in the *Window Manager* part of this manual.

Similarly, small, transiently loaded utilities may not require any memory over the 1024 bytes they are automatically allocated. Some programs and modules, however, will require arbitrary amounts of memory, which can be freed after use. For example, filing systems, specialised VDU drivers such as the font manager and so on. The memory manager provides simple allocation and deallocation facilities. Relocatable modules can use this manager either directly, to manipulate their own private heap, or indirectly using the module support calls.

A block of memory can be set up as a heap. This is a structure that allows arbitrary parts of the block to be allocated and freed. A program simply requests a block of a given size and is given a pointer to it by the heap manager. This block can be expanded or contracted or freed by using this pointer as a reference.

The part of the screen RAM that is not visible on the screen is also available as a temporary buffer. This memory is temporarily available because of the way that vertical scrolling is done.

One of the other memory resources available is the battery-backed CMOS RAM. This is used to hold default system parameters while the power is off. Modules, applications and users may use spare locations in CMOS RAM for their own purposes.

The MEMC chip controls how logical addresses (those used by programs or modules) are mapped into the physical memory location to use. Numerous calls are used to control how it does this, though generally this is something that most programs would not want to do.

Overview

Heap manager

RISC OS contains a heap management system. This is used by the operating system to allocate space within the relocatable module area and also to maintain the system heap. A heap is just an area of memory from which bytes may be allocated, then deallocated for later use. An area can also be reallocated, meaning that its size changes.

The heap manager is also available to the user. You provide an area of memory which is to be used for the heap, which can be any size you require. If you are a module, then the heap would be a block within the RMA, and if you are a program, then it would be within the application space.

Thus, it would be a heap within a heap. ie. A block in the RMA, for example, would be allocated by a module and then declared as a heap. In theory, this process could continue indefinitely, but in practice this is as far as you need to go.

At the start of a heap, the heap manager sets up the heap descriptor, which is a block containing information on the limits of the heap, etc. This descriptor is updated by the heap manager when necessary.

When a block within this heap is required, a request is made to the heap manager, which returns a pointer to a suitable block of memory. The heap manager keeps a record of the total amount of memory which is free in the heap and the largest individual block which is available.

Heap fragmentation

The heap management system does not provide garbage collection. This is the technique of moving blocks of allocated memory around so as to maximise the contiguous free space and avoiding excessive fragmentation of the heap.

Also, the heap management system will never attempt to move a block within the heap, since it has no knowledge of whether the block contains pointers that need to be relocated, or whether there are any pointers to the block which need updating. Hence, unless an area of contiguous free space of the size requested is available, a request for a block will fail.

MEMC control

MEMC maps logical onto physical addresses. To do this, it maintains a table of 128 entries that map a given memory block to a particular address. Generally, the system will take care of the operation of this mapping for you. Calls are provided to allow you to read this mapping and alter it, but you should have a very good reason to do so, and be certain of what you are doing.

Screen memory

The vertical scrolling technique used under RISC OS is to change the memory location that the screen starts at. This means that part of the screen memory may be unused, depending on the screen mode and the amount of memory reserved. You can use this memory temporarily, as long as you don't cause any output that may scroll the screen. Also remember that this memory is limited to one program using it at a time, so it may not be available every time you request it. Consequently, you cannot count on it being there when writing a module or application.

Battery-backed CMOS RAM

A block of 240 bytes of battery-backed CMOS RAM is available under RISC OS. Each location has a specific meaning and should not be directly modifed unless you are sure of the meaning of the value. Many of these locations are changed indirectly using the *Configure commands. These can be found throughout this manual, in the chapter appropriate to their function.

Some bytes are not allocated, and are reserved for users and applications to use. If you want to use one or more of the application bytes, you should request a location in writing from Acorn Computers. This is so that different applications don't accidentally use the same location.

Technical Details

Heap Manager

The heap is controlled by a single SWI, OS_Heap (SWI &1D). This has a reason code and can perform the following operations:

Reason code	Meaning
0	Initialise heap
1	Describe heap
2	Allocate a block from a heap
3	Free a block
4	Change the size of a block
5	Change the size of a heap
6	Read the size of a block

Internal format of the heap

A description of the structure used by the heap manager is given below. It should be noted that this structure is not guaranteed to be preserved between releases of the software and should not be relied upon. It is given purely for advanced programmers who may want to interpret the current state of the heap when testing and debugging their own code.

The heap descriptor is a block of four words:

&00	Special heap word
&04	Free list offset
&08	Heap base offset
&0C	Heap end offset

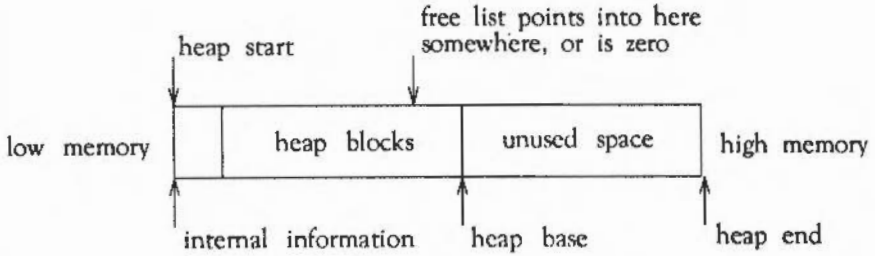
The 'special' heap word contains a pattern which distinguishes correct heap descriptors. The pattern is made up of the characters 'Heap' – which is &70616548 in hex.

All other words are offsets into the heap. This means that the heap is relocatable unless you place non-relocatable information in it.

The free list offset is an offset to the first free block in the heap, or zero if there are no free blocks. If the word is non-zero, the first free block is at address:

heap start + free list offset + 4

The other entries are offsets from the start of the heap which refer to boundaries within the heap structure. The heap is delimited as follows:



Blocks in the free list have information in the first two words as follows:

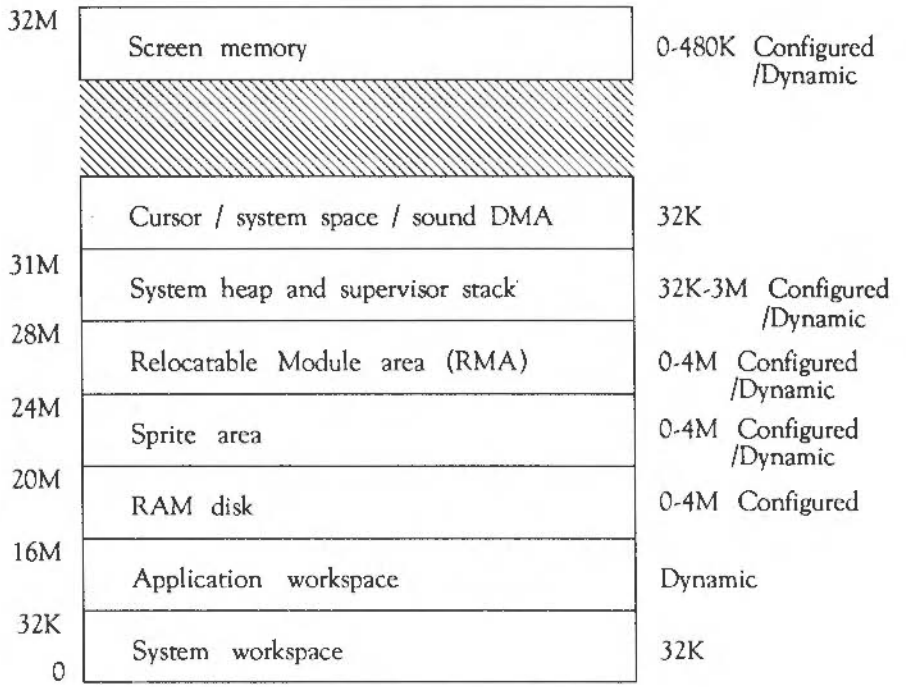
- Word 0 is the link to the next free block or 0 if at the end
- Word 1 is the size of this block (including these two words)

Allocated blocks start with a word which holds the size of the allocated block. The pointer returned by SWI OS_Heap when a block is allocated actually points to the second word which is the start of the memory available.

Allocation forces the block size to be a multiple of eight, to ensure that no matter what you do, the fragments can always be freed. Therefore, the minimum size of area that can be initialised is 24 bytes (16 for the fixed information and 8 for a block).

Logical memory map

The organisation of the logical address space is currently as follows:



You must not assume that any of the above addresses will remain fixed (save for the base of application workspace). There are defined calls to read any addresses you need, and you must use them.

Setting up the memory map

The memory map is set up on hard reset as follows:

- The permanent 32K allocations for system workspace at addresses &0000000 and &1F00000 (31Mbytes) are made, as well as some other fixed allocations (such as an initial part of the system heap).
- Then space is allocated to the various adjustable size regions, such as the screen., the system heap, the RMA, etc. Some of these have an absolute configured size, such as the screen. This is allocated in full. For other regions (such as the system heap and RMA), the configured size is the amount of free space that will be left; these only have a minimal allocation made at this stage.

- The rest of mermory is then allocated to the application workspace, from address &8000 up.
- System ROM and expansion card modules are then initialised.
- Finally, the regions that have a configured free space get allocated. First they are shrunk as far as possible (to ensure as close to 0 bytes free as possible), then a block of the configured size is requested and freed, so that the heaps contain as close to the configured free space as possible.

Example memory allocation

Here is an example of how memory might be allocated given some typical RAM size allocations on an A310 (8K page size):

Area	Pages	Page size	Total
FontSize	20	4K	80K
RamFsSize	0	8K	0
RMASize	16	8K	128K
ScreenSize	20	8K	160K
SpriteSize	10	8K	80K
SystemSize	4	8K	32K+32K
System workspace			32K
Cursor etc. workspace			32K
Total	576K		
Application area	1024K – 576K = 448K		

A configured screen size of 0 means 'default for this machine', which is 160K on an A310 (see *Configure ScreenSize).

As outlined above, the size of the system area (at 28M) is shrunk as far as possible after all module initialisation and then 'n' extra pages are added. 8K of this is used for the system stack. The rest is for OS variable storage (eg alias variables) and module information. The configured amount is added to the 32K initially allocated.

Altering the memory map

While no application is running (ie in the supervisor prompt), the memory map can be altered as required. For example, if you load a module from disc and the RMA isn't big enough to hold it, the size of the RMA will be increased by an appropriate amount. The OS can only do this when there is no

application active, as the extra memory has to be taken from the application workspace. Most programs don't react too kindly to large areas of their memory allocation disappearing.

Under an environment such as the Wimp desktop, multiple applications are run concurrently. The currently running application is mapped into &8000. When the Wimp decides to swap to another application, it maps the current one out and maps the new application into that space. Thus, every application is given the illusion that it is the only one in the system. Before each call your application makes to Wimp_Poll (which is when it may be swapped out), it must call OS_DelinkApplication (SWI &4D) to remove any vectors that point into the application area – if it has any to remove, that is. When its call to Wimp_Poll returns (and hence it is swapped back in), it must then call OS_RelinkApplication (SWI &4E) to reload these vectors.

Page size

The SWI OS_ReadMemMapInfo (SWI &51) returns the pagesize used in the system and the number of pages present. For more details of page sizes, see the chapter entitled *ARM hardware*.

Controlling memory allocation

OS_ChangeDynamicArea (SWI &2A) allows control of the space allocated to the system heap, RMA, screen, sprite area, font cache and RAM filing system. Any space left over is the application space by default. Any of these settings can be read with OS_ReadDynamicArea (SWI &5C). OS_ReadRAMFsLimits (SWI &4A) will read the range of bytes used by the RAM filing system. The size of it can be set in CMOS RAM using *Configure RamFsSize. See also *Configure RMASize and *Configure SystemSize.

Memory protection

You have read/write access to much of the logically mapped RAM. There are exceptions, such as the 32K system workspace at &1F00000 (31M), the RAMdisc, and the font cache. More areas may become protected in future releases of RISC OS. The **only** areas you should directly access are the application workspace and the RMA. It is very dangerous to write to **any** other areas, or rely on certain locations containing given information, as these are subject to change. You should always use OS routines to access operating system workspace.

OS_ValidateAddress (SWI &3A) will check a range of logical addresses to see if they are mapped into physical memory.

Changing the logical map

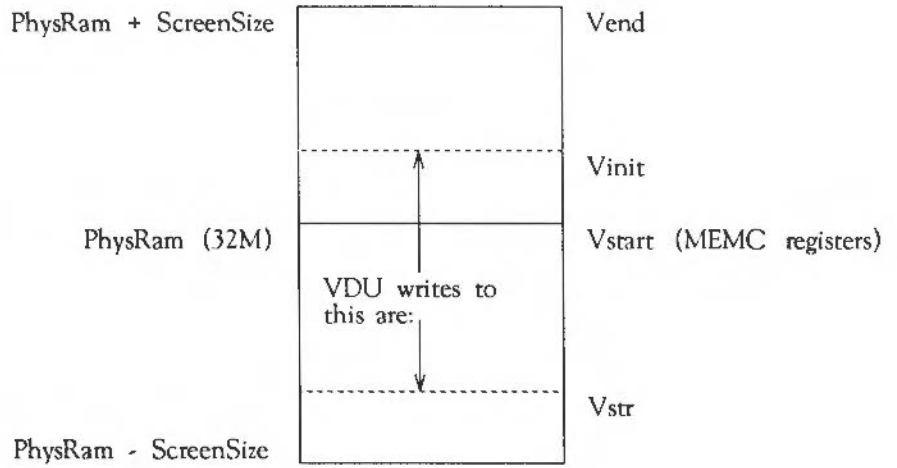
The mapping that MEMC maintains from logical to physical address space can be read with OS_ReadMemMapEntries (SWI &52). This gives a list of physical addresses for a matching set of logical page numbers.

The reverse operation, OS_SetMemMapEntries (SWI &53) will write the mapping inside MEMC. Note that this is an extremely dangerous operation if you are not sure what you are doing.

OS_UpdateMEMC (SWI &1A) is a lower level operation that alters the bits in the MEMC control register.

Screen memory

Hardware scrolling is implemented by having the screen workspace at the end of logical memory, adjacent to the corresponding physical RAM banks which are mapped onto those addresses. This means that there are two adjacent copies of the screen memory as follows:



The screen can, therefore, be scrolled vertically by altering the VDU driver screen start address as shown above. This is usually performed automatically and you don't have to concern yourself with it.

OS_ClaimScreenMemory (SWI &41) allows you to claim or release this space.

The screen-size is configurable in units of one page (8K or 32K). Hence for a 20K screen on a 400 series machine, 32K will have to be used since it is the next highest multiple of 32K. For an 80K screen, 96K would be used, etc. In addition, if you want to use multiple banks of screen memory (eg for animation), enough memory must be reserved for each bank.

Because the total screen memory is often much less than what is required at a given time, a facility is available whereby the 'extra' RAM can be claimed for short periods. It can be used as a buffer, in a data transfer operation, for example.

Non-volatile memory (CMOS RAM)

240 bytes of non-volatile memory are provided. Some of these are reserved since they hold default values for certain parameters and are set using various *Configure options. OS_Byte 161 allows you to read the CMOS memory directly, while OS_Byte 162 can write to it. The full list is given below:

Location	Function
0	Econet station number (not directly configurable)
1	Econet file server station id (0 => name configured)
2	Econet file server net number (or first char of name)
3	Econet printer server station id (0 => name configured)
4	Econet printer server net number (or first char of name)
5	Default filing system number
6 - 9	Reserved for Acorn use
10	Screen info: Bits 0 - 3 screen mode number. This is held in 5 bits The fifth bit is bit 1 in byte 133 Bit 4 TV interlace (first *TV parameter) Bits 5 - 7 TV vertical adjust (signed three-bit number)
11	Shift, Caps mode: Bits 0 - 2 reserved Bits 3 - 5 ShCaps (001), NoCaps (010), Caps (100) Bit 6 - 7 reserved
12	Keyboard auto-repeat delay
13	Keyboard auto-repeat rate
14	Printer ignore character
15	Printer information: Bit 0 reserved Bit 1 0 => Ignore, 1 => NoIgnore

	Bits 2 - 4 serial baud rate (0=75,...,7=19200)
	Bits 5 - 7 printer type
16	Miscellaneous flags
	Bit 0 reserved
	Bit 1 0 => Quiet, 1 => Loud
	Bit 2 reserved
	Bit 3 0 => Scroll, 1 => NoScroll
	Bit 4 0 => NoBoot, 1 => Boot
	Bits 5 - 7 serial data format (0..7)
17 - 29	Reserved for Acorn use
30 - 45	Reserved for the user
46 - 79	Reserved for applications
80 - 111	Reserved for RISC iX
112 - 127	Reserved for expansion card use
128 - 129	Current year
130	Reserved for Acorn use
131	Reserved for Acorn use
132	DumpFormat
	Bits 0,1 control character print control
	00 print in GSTrans format
	01 print as a dot
	10 print decimal inside angle brackets
	11 print hex inside angle brackets
	Bit 2 treat top-bit-set characters as valid if set
	Bit 3 AND character with &7F in *Dump
	Bit 4 treat TAB as print 8 spaces
	Bit 5 Tube expansion card enable
	Bits 6,7 Tube expansion card slot (0 - 3)
133	Sync, monitor type, some mode information
	Bit 0 SyncBit
	Bit 1 top bit of mode configuration number in byte 10
	Bits 2 - 3 monitor type
134	Fontsize in units of 4K
135 - 137	ADFS use
138 - 139	Set *Cat format
140 - 141	Set *Examine format
142	Twin's byte
143	Screen size in pagesize units.
144	RAM disc size in pagesize units
145	System heap size in pagesize to add after initialisation

146	RMA size in pagesize to add after initialisation	
147	Sprite size in pagesize	
148	SoundDefault parameters	
	Bits 0 - 3	channel 0 default voice
	Bits 4 - 6	loudness (0 - 7 => &01, 13, 25, 37, 49, 5B, 6D, 7F)
	Bit 7	loudspeaker enable
149 - 152	BASIC Editor	
153 - 157	Printer server name	
158 - 172	File server name	
173 - 176	*Unplug for ROM modules. 32 bits for up to 32 modules	
177 - 180	4 * 8 bits for unplugged modules in expansion cards	
181 - 184	Wild card for BASIC editor	
185	Configured language	
186	Configured country	
187	VFS	
188	Bottom 2 bits are ROMFS Opt 4 state	
189	Winchester size	
190	Protection state	
	Bit 0	Peek
	Bit 1	Poke
	Bit 2	JSR
	Bit 3	User RPC
	Bit 4	OS RPC
	Bit 5	Halt
	Bit 6	GetRegs
191	Mouse multiplier	
192	System speed – currently unused	
	Bits 0 - 3	RAM speed
	Bit 4	ROM speed
	Bit 5	Cache enable for ARM3
193	Wimp mode	
194	Wimp flags	
195	Desktop state	
	Bits 0,1	display mode (Filer)
	00	large icons
	01	small icons
	10	full info
	11	reserved
	Bits 2,3	sorting mode (Filer)
	00	sort by name

	01	sort by type
	10	sort by size
	11	sort by date
	Bit 4	sorting mode (0=name, 1=number)
	Bit 5	confirm option (1=confirm)
	Bit 6	verbose option (1=verbose)
	Bit 7	reserved
196	ADFS dir cache	
197 - 204	FontMax, FontMax1 - FontMax7	
205 - 211	Reserved for RISC iX	
224 - 238	Reserved for RISC iX	
239	One byte for CMOS RAM checksum; not used currently	

OS_Byte 161 (SWI &06)

Read battery-backed CMOS RAM

On entry

R0 = 161 (&A1) (reason code)
R1 = RAM location

On exit

R0, R1 preserved
R2 = contents of location

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call provides read access to any of the locations in the battery-backed CMOS RAM. For example, this call may be used by a module to read a default configuration parameter. Moreover, this parameter could be examined by the user using the *Status command, if the module provides a suitable entry in its command decoding table. See the chapter entitled *Modules* for more details.

Related SWIs

OS_Byte 162 (SWI &06)

Related vectors

ByteV

OS_Byte 162 (SWI &06)

Write battery-backed CMOS RAM

On entry

R0 = 162 (&A2) (reason code)
R1 = RAM location
R2 = value to be written

On exit

R0, R1 preserved
R2 = corrupted

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call provides write access to any of the locations in the battery backed RAM with the exception of location zero, which is protected.

Related SWIs

OS_Byte 161 (SWI &06)

Related vectors

ByteV

OS_UpdateMEMC (SWI &1A)

Read or alter the contents of the MEMC control register

On entry

R0 = new bits in field
R1 = field mask

On exit

R0 = previous bits in field
R1 = previous field mask

Interrupts

Interrupts are disabled
Fast interrupts are disabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entered because interrupts are disabled

Use

The memory controller (MEMC) chip is a write-only device. The operating system maintains a software copy of the current state of the control register and OS_UpdateMEMC updates MEMC from the software state. To allow the programming of individual bits the call takes a field and a mask. The new MEMC value is:

```
newMemC  = (oldMEMC AND NOT R1) OR (R0 AND R1)
R0       = oldMEMC
```

So to read the contents without altering them, R1 and R2 should both be zero. To set them to 'n', R1=&FFFFFFFF and R2=n.

Related SWIs

None

Related vectors

None

OS_Heap 0 (SWI &1D)

Initialise Heap

On entry

R0 = 0 (reason code)
R1 = pointer to heap to initialise
R3 = size of heap

On exit

R0, R1, R3 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call checks the given heap pointer, and then writes a valid descriptor into the heap it points at. The heap is then ready for use. The value given for R1 must be word-aligned and less than 32Mbytes (ie must point to an area of logical RAM). R3 must be a multiple of four and less than 16Mbytes.

Related SWIs

None

Related vectors

None

OS_Heap 1 (SWI &1D)

Describe Heap

On entry

R0 = 1 (reason code)
R1 = pointer to heap

On exit

R0, R1 preserved
R2 = largest available block size
R3 = total free

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call returns information on the space available in the heap. An error is returned if the heap is invalid. This may be for any of the following reasons:

- the heap descriptor is corrupt
- the information within the heap is not sensible
- R1 does not point to a heap

Related SWIs

None

Related vectors

None

OS_Heap 2 (SWI &1D)

Get heap block

On entry

R0 = 2 (reason code)
R1 = pointer to heap
R3 = size required in bytes

On exit

R0, R1 preserved
R2 = pointer to claimed block or zero if allocation failed
R3 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This allocates a block from the heap. An error is returned if the allocation failed for any of the following reasons:

- there is not a large enough block left in the heap
- the heap has been corrupted
- R1 does not point to a heap

Related SWIs

None

Related vectors

None

OS_Heap 3 (SWI &1D)

Free heap block

On entry

R0 = 3 (reason code)
R1 = pointer to heap
R2 = pointer to block

On exit

R0 - R2 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This checks that the pointer given refers to an allocated block in the heap, and deallocates it. Deallocation tries to join free blocks together if at all possible, but if the block being freed is not adjacent to any other free block it is just added to the list of free blocks. An error is returned if the deallocation failed which may be because:

- R1 does not point to a heap
- the heap descriptor or heap was corrupted
- R2 does not point to an allocated block in the heap.

Related SWIs

None

Related vectors

None

OS_Heap 4 (SWI &1D)

Extend heap block

On entry

R0 = 4 (reason code)
R1 = pointer to heap
R2 = pointer to block
R3 = required size change in bytes (signed integer)

On exit

R0, R1 preserved
R2 = new block pointer
R3 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This attempts to enlarge or shrink the given block in its current position if possible, or, if this is not possible, by reallocating and copying it. Note that if the block has to be moved, it is your responsibility to note this (by the fact that R2 has been altered), and to perform any necessary relocation of data within the block.

Related SWIs

None

Related vectors

None

OS_Heap 5 (SWI &1D)

Extend heap

On entry

R0 = 5 (reason code)
R1 = pointer to heap
R3 = required size change in bytes (signed integer)

On exit

R0, R1, R3 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This updates the heap size information to take account of the new size. An error is returned if it cannot shrink far enough, because of data that has already been allocated.

Related SWIs

None

Related vectors

None

OS_Heap 6 (SWI &1D)

Read block size

On entry

R0 = 6 (reason code)
R1 = pointer to heap
R2 = pointer to block

On exit

R0 - R2 preserved
R3 = current block size

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This reads the size of a block in the specified heap. An error is returned if the heap or the block could not be found.

Related SWIs

None

Related vectors

None

OS_ChangeDynamicArea (SWI &2A)

Alter the space allocation of a dynamic area

On entry

R0 = area to alter
R1 = amount to move in bytes (signed integer)

On exit

R0 = preserved
R1 = number of bytes being given to application
(ie –ve numbers ⇒ application shrinking, so area is growing)

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

OS_ChangeDynamicArea allows the space allocated to an area to be altered in size by removing or adding workspace from the application workspace.

The area to be altered depends on R0 as follows:

Value of R0	Area to alter
0	system heap
1	RMA
2	screen area
3	sprite area
4	font cache
5	RAM filing system

The amount to move is given by the sign and magnitude of R1:

+ve means **shrink** the selected area
–ve means **enlarge** the selected area

Note that normally, this cannot be used while the application work area is being used; for example when a language is active outside the RISC OS desktop. An attempt to do so will result in a `Memory in use` error. (In fact,

when this call is made, the OS passes a service call round to modules, which can veto the change if they can't handle it correctly. See the chapter entitled *Modules* for more details.)

Any area size change will fail if the new size is smaller than the current requirements, but will shrink the area as far as it can. If you need to release as much space as possible from an area, try to reduce its size by 16 Mbytes.

Expanding, on the other hand, does nothing if it can't move enough. In this case, if you asked for the extra space you probably need it all; RISC OS assumes that half the job is no use to you.

This SWI also does an upcall, to enable programs running in application workspace to allow movement of memory. If the upcall is claimed when the application is running in application workspace, the memory movement is allowed to proceed. R1 is the amount of memory that is being removed from the application workspace. R0 in the service call contains the amount you are attempting to move.

An error is returned if not all the bytes were moved, or if application workspace is being used – ie an application is active.

Related SWIs

OS_ReadDynamicArea (SWI &5C)

Related vectors

None

OS_ValidateAddress (SWI &3A)

Check that a range of addresses are in logical RAM

On entry

R0 = minimum address
R1 = maximum address

On exit

R0, R1 preserved
C flag is clear if the range is OK, set otherwise

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI checks the address range between R0 and R1 minus 1 to see if they are valid. If they are equal, then that single address is checked. Valid addresses are in logical RAM (0 - 32M) and have a mapping into physical RAM, including screen RAM, throughout the specified range.

Related SWIs

None

Related vectors

None

OS_ClaimScreenMemory (SWI &41)

Use spare screen memory

On entry

R0 = 0 for release, 1 for claim
R1 = length required in bytes (if R0 = 1)

On exit

R0 = preserved
if the C flag is 0, then memory was claimed successfully
    R1 = length available
    R2 = start address
if the C flag is 1, then memory could not be claimed
    R1 = length that is available

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

There are several restrictions to the use of screen memory. It can only be claimed by one 'client' at a time, who gets all of it. It can only be claimed if no bank other than bank 1 has been used. You can't claim it, for example, if the shadow bank has been used.

While you have claimed the screen memory, you must not perform any action which might causes the screen to scroll. This means avoiding the use of routines which might cause screen output.

It is important to release the memory after it has been used.

Related SWIs

None

Related vectors

None

OS_ReadRAMFsLimits (SWI &4A)

Get the current limits of the RAM filing system

On entry

—

On exit

R0 = start address
R1 = end address + 1 byte

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This reads the start and end addresses of the RAM filing system. This information can also be read from OS_ReadDynamicArea.

If the RamFS is configured to zero size then R0 and R1 have the same value on exit.

The size of the RamFS after a hard reset (ie the difference between the two return values) can be configured using *Configure RamFsSize.

Related SWIs

OS_ReadDynamicArea (SWI &5C)

Related vectors

None

OS_DelinkApplication (SWI &4D)

Remove any vectors that an application is using

On entry

R0 = pointer to buffer
R1 = buffer size in bytes

On exit

R0 preserved
R1 = number of bytes left in buffer

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI cannot be re-entrant because interrupts are disabled

Use

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

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

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

Related SWIs

OS_RelinkApplication (SWI &4E)

Related vectors

None

OS_RelinkApplication (SWI &4E)

Restore any vectors that an application is using from a buffer

On entry

R0 = pointer to buffer

On exit

R0 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

When an application is going to be swapped in, all vectors that it uses must be restored.

R0 on entry points to a buffer, which has previously been created by OS_DelinkApplication .

Related SWIs

OS_DelinkApplication (SWI &4D)

Related vectors

None

OS_ReadMemMapInfo (SWI &51)

Read the page size and count

On entry

—

On exit

R0 = page size in bytes
R1 = number of pages

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the page size used by MEMC and the number of pages in use. The valid page numbers are 0 to R1 - 1, and the total memory size is R0 times R1 bytes.

Related SWIs

None

Related vectors

None

OS_ReadMemMapEntries (SWI &52)

Read the logical to physical memory mapping used by MEMC

On entry

R0 = pointer to request list

On exit

R0 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

The request list is a series of entries three words long, terminted by a –1 in the first word. The three words are used for:

word 1 : page number (set on entry)
word 2 : address page that is is mapped to (set in SWI)
word 3 : protection level. This is a bitfield, which uses the bottom 2 bits

00	readable and writable by everybody
01	read-only in user mode
10	inaccessible in user mode

all other bits are reserved and must be written as zero.

Related SWIs

OS_SetMemMapEntries (SWI &53)

Related vectors

None

OS_SetMemMapEntries (SWI &53)

Write the logical to physical memory mapping used by MEMC

On entry

R0 = pointer to request list

On exit

R0 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

The request list is a series of entries three words long, terminted by a –1 in the first word. This is described in OS_ReadMemMapEntries. It contains all fields set on entry.

Any address above 32Mbyte (&2000000) makes that page inaccessible. This also sets the protection level to minimum accessibility.

This SWI assumes you know what you are doing. It will set any page to any address, with no checks at all.

If you are using this call, then you can only use OS_ChangeDynamicArea if the kernel's limits are maintained, and all appropriate areas contain continuous memory..

Related SWIs

OS_SetMemMapEntries (SWI &53), OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

OS_ReadDynamicArea (SWI &5C)

Read the space allocation of a dynamic area

On entry

R0 = area to read

On exit

R0 corrupted
R1 = current number of bytes in area

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI reads the size of an area. The area read depends on R0 as follows:

Value of R0	Area to read
0	system heap
1	RMA
2	screen area
3	sprite area
4	font cache
5	RAM filing system

Related SWIs

OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

*Configure

Set a parameter in the CMOS RAM

Syntax

```
*Configure [<param1>[<param2>]]
```

Parameters

<param1>	setting to be changed
<param2>	new value

Use

*Configure defines the configuration settings held in the CMOS RAM. These are made current on initial power-on and after a hard break (Ctrl RESET), and do NOT take effect immediately.

If the command is given with no parameters at all, then the configuration options are listed.

If it is given with parameters then it is used to alter a particular setting. <param~1> identifies the setting to be changed, as defined below. <param 2> defines the value to be stored in the appropriate location in the CMOS RAM. Some settings have more than one <param 2>, and sometimes there are none at all.

Where a number is required, it may be given in decimal, as a hex number preceded by &, or a number of the form base_num, where base is the base of the number in decimal in the range 2 to 36. For example 2_1010 is another way of saying 10.

Here is a list of the available configure parameters. The details of each command can be found in the appropriate chapter:

User Preferences	In the chapter
*Configure Boot	FileSwitch
*Configure Caps	Character input
*Configure Delay	Character input
*Configure Dir	FileSwitch
*Configure DumpFormat	FileSwitch
*Configure FileSystem	FileSwitch
*Configure Language	The rest of the kernel
*Configure Lib	FileSwitch
*Configure Loud	VDU drivers
*Configure Mode	VDU drivers

*Configure MouseStep	VDU drivers
*Configure NoBoot	FileSwitch
*Configure NoCaps	Character input
*Configure NoDir	FileSwitch
*Configure NoScroll	VDU drivers
*Configure Quiet	VDU drivers
*Configure Repeat	Character input
*Configure Scroll	VDU drivers
*Configure ShCaps	Character input
*Configure SoundDefault	The Sound system
*Configure WimpFlags	The Window Manager
*Configure WimpMode	The Window Manager

Hardware configuration	In the chapter
*Configure Baud	Character output
*Configure Country	International module
*Configure Data	Character output
*Configure Drive	ADFS
*Configure Floppies	ADFS
*Configure FS	NetFS
*Configure HardDiscs	ADFS
*Configure Ignore	Character output
*Configure MonitorType	VDU drivers
*Configure Print	Character output
*Configure PS	NetPrint
*Configure Step	ADFS
*Configure Sync	VDU drivers
*Configure TV	VDU drivers

Memory allocation	In the chapter
*Configure ADFSbuffers	ADFS
*Configure ADFSDirCache	ADFS
*Configure FontSize	Font manager
*Configure RAMFsSize	Memory management
*Configure RMASize	Memory management
*Configure ScreenSize	VDU drivers
*Configure SpriteSize	Sprites
*Configure SystemSize	Memory management

Example	`*Configure Baud 7`
Related commands	*Status
Related SWIs	None
Related vectors	None

*Configure RamFsSize

Sets the size of memory for the RAM filing system

Syntax

```
*Configure RamFSSize <n>|<mK>
```

Parameters

<n>	number of pages of memory; n ≤ 127
<mK>	kilobytes of memory reserved

Use

*Configure RamFSSize sets the size of memory that will be used for the RAM Filing System (when the RAMFS module is present) after the next hard reset. The default value is 0, which disables the RAM filing system.

Example

```
*Configure RamFSSize 128K
```

Related commands

None

Related SWIs

OS_ReadRAMFsLimits (SWI &4A), OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

*Configure RMASize

Reserves an extra area of memory for relocatable modules

Syntax

```
*Configure RMASize <n>|<mK>
```

Parameters

<n>	number of pages of memory; n <= 255
<mK>	kilobytes of memory reserved

Use

*Configure RMASize is used to reserve an extra area of memory in the relocatable module area (RMA) after all modules have been initialised. If <n> = 0, no extra memory is reserved. The default is to reserve 2 extra pages.

Example

```
*Configure RMASize 128K
```

Related commands

None

Related SWIs

OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

*Configure SystemSize

Reserves an extra area of RAM for the system heap

Syntax

```
*Configure SystemSize <n>|<mK>
```

Parameters

<n>	number of pages of memory; n <= 63
<mK>	kilobytes of memory reserved

Use

*Configure SystemSize reserves an extra area of memory for the system heap after all modules have been initialised. The default value is 0.

Example

```
*Configure SystemSize 32K
```

Related commands

None

Related SWIs

OS_ChangeDynamicArea (SWI &2A)

Related vectors

None

*Status

Provides information on how the computer is configured

Syntax

`*Status [<name>]`

Parameters

`<name>` a *Configure option name

Use

`*Status` displays the configuration status of the computer. Because the default values of various parameters are held in non-volatile memory (the battery-backed CMOS RAM), they are preserved even when the computer is switched off, until reset from the desktop (by using Configure) or by using the *Configure command.

`*Status` without a parameter list displays all values.

`*Status <name>` displays the value of the *Configure option name.

Example

*Status TV

Related commands

*Configure

Related SWIs

None

Related vectors

None

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-redundency 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

OS_Byte 0 (SWI &06)

Display OS version information

On entry

R0 = 0 (reason code)
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.

This command (display message) can also be performed by `*FX 0`

Related SWIs

None

Related vectors

ByteV

OS_Byte 1 (SWI &06)

Write user flag

On entry

R0 = 1 (reason code)
R1 = new value

On exit

R0 preserved
R1 = old 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.

This command can also be performed by `*FX 1,<value>`

Related SWIs

OS_Byte 241 (SWI &06)

Related vectors

ByteV

OS_Byte 241 (SWI &06)

Read/write user flag

On entry

R0 = 241 (&F1) (reason code)
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.

The write command can also be performed by `*FX 241,<value>`

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 bit29 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 Massachusets, pages 145 -149.

It is not as fast as a quick sort 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 16Kbytes 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, the 'Y' key inidcates yes and any other key, no.

The result is returned in lowercase, irrespective of the keyboard state.

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 => read, bit 0 =1 => write)
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

*Configure Language

Selects the language used at power on

Syntax

```
*Configure Language <n>
```

Parameters

<n>	the module number of the language which will be started after power on. The default is n=3 (the desktop).

Use

*Configure Language followed by a number sets the module language which is automatically selected on power on. Choose 3 for the desktop, 4 for BASIC or 0 for Command Line mode. Use the *Modules command to check the number of the required language, especially if you have added or removed modules.

Note that the Desktop exits immediately if a boot file is run at power on using *Exec. Instead, you should select the Desktop 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 information about each command

Syntax

```
*Help [<keyword>]
```

Parameters

<keyword> the command name(s) to get help on

Use

The *Help command gives brief information about each command in the machine operating system. *Help <keyword> displays a brief explanation of the keyword and, if the keyword is a command name, the syntax of the command.

If you are not sure about the name of a command:

`*Help Commands`	will list all the available utility commands;
`*Help FileCommands`	will list all the commands relating to filing systems;
`*Help Modules`	will list the names of all currently loaded modules, with their version numbers and creation dates;
`*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 screenfull 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