Software vectors

Introduction

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

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

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

Overview

Claiming vectors

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

You can change this address by using the *SWI OS_Claim (on page 8)*, documented later in this chapter. RISC OS will then instead branch to your own routine, held at the address you pass to OS_Claim.

Your own routine can do one of the following:

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

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

An example

As an example, let's look at the *SWI OS_WriteC (on page 0)* routine. When RISC OS decodes a SWI with SWI number &00, it knows that you are requesting a write character operation. RISC OS gets an address from a vector - in this case called *WrchV (on page 0)* - and passes control to the routine.

Now by default, the WrchV contains the address of the standard write character routine in ROM. If you claim the vector using *SWI OS_Claim (on page 8)*, whenever an OS_WriteC is executed, your own routine will be called first.

Vector chains

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

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

The routines are called in the reverse order to the order in which they called *SWI OS_Claim (on page 8)*. The last routine to OS_Claim the vector will be the first one called. If that routine passes the call on, the next most recent claimant will get the call, and so on. If any of the routines on the vector intercept the call, the earlier claimants will not be called.

When not to intercept a vector

There are some vectors which should not be intercepted; they must always be passed on to other claimants. This is because the default owner, ie the routine which is called if no one has claimed the vector, might perform some important action. The error vector, *ErrorV (on page 0)*, is a good example. The default owner of this vector is a routine which calls the error handler. If you intercept ErrorV, the error handler will never be called, and errors won't be dealt with properly.

Multiply installing the same routine

When *SWI OS_Claim (on page 8)* adds a routine to a vector, it automatically removes any identical earlier instances of the routine from the chain (ie instances having the same pointer to code, and the same pointer to workspace). If you don't want this to happen, use the *SWI OS_AddToVector (on page 12)* instead.

Desktop applications

Under an environment such as the desktop, multiple applications are run concurrently. The currently running application is mapped into memory at &8000. Desktop applications periodically return control to the Window Manager (or **Wimp**) by calling the *SWI Wimp_Poll (on page 0)*; at this point the Wimp may decide to swap to another application. In doing so, it maps the current application out of the application space, and maps the new application into that space. Thus every application is given the illusion that it is the only one in the system.

If your application has claimed a vector using a routine in its own space, it must obviously release that vector each time it (and the claiming routine) may be swapped out of application space. Before each call your application makes to Wimp_Poll (which is when it may be swapped out), it must call *SWI OS_DelinkApplication (on page 13)* to remove any claiming routines in application space. When its call to Wimp_Poll returns (and hence it is swapped back in), it must then call *SWI OS_RelinkApplication (on page 14)* to reclaim those vectors.

Technical details

Use of registers

If you write a routine that uses a vector, it must obey the same entry and exit conditions as the corresponding RISC OS routine. For example, a routine on *WrchV (on page 0)* must preserve all registers, just as the *SWI OS_WriteC (on page 0)* does.

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

Processor modes

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

- Routines vectored through *IrqV (on page 16)* are always executed in IRQ mode.
- Routines vectored through Vectors &10 to &16 (*EventV (on page 0)*, *InsV (on page 17)*, *KeyV (on page 0)*, *RemV (on page 23)*, *CnpV (on page 29)*) and *TickerV (on page 32)* are generally executed in IRQ mode, but may be executed in SVC mode if called using *SWI OS_CallAVector (on page 11)*, and in certain other unspecified circumstances.
- All other routines are executed in SVC mode – the mode entered when the SWI instruction is executed.

SVC mode

Note that if you call a SWI from a routine that is in SVC mode, you will corrupt the return address held in R14. Consequently, your routine should use the full, descending stack addressed by R13 to save R14 first. See the section entitled *Important notes (on page 0)* for a more complete explanation of this.

IRQ mode

If your routine will be entered in IRQ mode there are other restrictions. These are detailed in full in the *Restrictions (on page 0)*.

Returning errors

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

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

Returning from a vectored routine

You should use one of two methods to return from a vectored routine. These are described immediately below; for an example, see the *example program (on page 36)*.

Passing on the call

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

```
MOVS PC,R14
```

Intercepting the call

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

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

Control will return to the caller of the vector.

More complex uses of vectors

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

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

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

- the vector still looks exactly the same to a program that is calling it, even if it now does different things
- your routine will cope with being called in all the processor modes that its vector uses (for example, SVC or IRQ mode for a routine on *InsV (on page 17)*)
- the values of R10 and R11 are preserved when earlier claimants of the vector are repeatedly called.

Vector defintions

In most cases, the interrupt status is given as undefined. This is because the vectors may be called either by the SWI(s) which normally use them, many of which ensure a given interrupt status, or by *SWI OS_CallAVector (on page 11)*, which does not alter the interrupt status.

List of software vectors

The software vectors are listed below. The names of the routines which can cause each vector to be called are in brackets:

Number	Vector	Description
&00	UserV	*User vector (on page 15)* is reserved and must not be used
&01	ErrorV	*Error vector (on page 0)* (*SWI OS_GenerateError (on page 0)*)
&02	IrqV	*Unknown interrupt vector (on page 16)*
&03	WrchV	*Write character vector (on page 0)* (*SWI OS_WriteC (on page 0)*)
&04	RdchV	*Read character vector (on page 0)* (*SWI OS_ReadC (on page 0)*)
&05	CLIV	*Command line interpreter vector (on page 0)* (*SWI OS_CLI (on page 0)*)
&06	ByteV	*OS_Byte indirection vector (on page 0)* (*SWI OS_Byte (on page 0)*)
&07	WordV	*OS_Word indirection vector (on page 0)* (*SWI OS_Word (on page 0)*)
&08	FileV	*File read/write vector (on page 0)* (*SWI OS_File (on page 0)*)
&09	ArgsV	*File arguments read/write vector (on page 0)* (*SWI OS_Args (on page 0)*)
&0A	BGetV	*File byte read vector (on page 0)* (*SWI OS_BGet (on page 0)*)
&0B	BPutV	*File byte put vector (on page 0)* (*SWI OS_BPut (on page 0)*)
&0C	GBPBV	*File byte block get/put vector (on page 0)* (*SWI OS_GBPB (on page 0)*)
&0D	FindV	*File open vector (on page 0)* (*SWI OS_Find (on page 0)*)
&0E	ReadLineV	*Read a line of text vector (on page 0)* (*SWI OS_ReadLine (on page 0)*)
&0F	FSCV	*Filing system control vector (on page 0)* (*SWI OS_FSControl (on page 0)*)
&10	EventV	*Event vector (on page 0)* (*SWI OS_GenerateEvent (on page 0)*)
&11		*Reserved*
&12		*Reserved*
&13	KeyV	*Key vector (on page 0)*
&14	InsV	*Buffer insert vector (on page 17)* (*SWI OS_Byte 138 (on page 0)*)
&15	RemV	*Buffer remove vector (on page 23)* (*SWI OS_Byte 145 (on page 0)*)
&16	CnpV	*Count/Flush Buffer vector (on page 29)* (*SWI OS_Byte 21 (on page 0)* & *SWI OS_Byte 152 (on page 0)*)
&17	UKVDU23V	*Unknown VDU23 vector (on page 0)* (*SWI OS_WriteC (on page 0)*)
&18	UKSWIV	*Unknown SWI vector (on page 31)*
&19	UKPLOTV	*Unknown VDU25 vector (on page 0)* (*SWI OS_Plot (on page 0)*)
&1A	MouseV	*Mouse vector (on page 0)* (*SWI OS_Mouse (on page 0)*)
&1B	VDUXV	*VDU vector (on page 0)* (*SWI OS_WriteC (on page 0)*)
&1C	TickerV	*100Hz vector (on page 32)*
&1D	UpcallV	*Warning vector (on page 0)* (*SWI OS_UpCall (on page 0)*)
&1E	ChangeEnvironmentV	*Environment change vector (on page 0)* (*SWI OS_ChangeEnvironment (on page 0)*)
&1F	SpriteV	*Sprite indirection vector (on page 0)* (*SWI OS_SpriteOp (on*

Number	Vector	Description
		page 0))
&20	DrawV	*Draw SWI vector (on page 33)* (all *Draw (on page 0)* SWI calls)
&21	EconetV	*Econet activity vector (on page 34)* (all *Econet (on page 0)* SWI calls)
&22	ColourV	*ColourTrans SWI vector (on page 0)* (all *ColourTrans (on page 0)* SWI calls)
&23	PaletteV	*Read/write palette vector (on page 0)*
&24	SerialV	*OS_SerialOp indirection vector (on page 0)* (*SWI OS_SerialOp (on page 0)*)
&25	FontV	Font manager
&26	PointerV	*Mouse drivers (on page 0)*
&27	TimeShareV	SkyNet
&28	LowPriorityEventV	For future expansion
&29	FastTickerV	Like TickerV, but faster (RISCOS Ltd)
&2A	GraphicsV	Graphics hardware abstraction
&2B	UnthreadV	High-priority callbacks
&2C	VideoV	Graphics abstraction (RISCOS Ltd)
&2D	SeriousErrorV	Handling of “serious errors” and exceptions
&3E	NVRAMV	NVRAM hardware abstraction (RISCOS Ltd)
&3F	RTCV	RTC hardware abstraction (RISCOS Ltd)

All other vectors are currently reserved.

Additional information on software vectors

Many of the vectors are by default used to indirect calls of SWIs, and so the routine they call is the same as that the SWI calls.

About the filing system vectors

Note that the filing system vectors FileV (Vector &08) to FindV (Vector &0D) have 'no default action', ie they return immediately. However, the *FileSwitch (on page 0)* module *SWI OS_Claim (on page 8)*s the vectors whenever the machine is reset, so effectively the default action is to perform the appropriate filing system routine.

Other vectors and resets

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

SWI Calls

OS_Claim (SWI &1F)

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

On entry

R0 = vector number (see *List of software vectors (on page 5)*)
R1 = address of claiming routine that is to be added to vector
R2 = value to be passed in R12 when the routine is called

On exit

R0 preserved
R1 preserved
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 adds the routine whose address is given in R1 to the list of routines claiming the vector. This becomes the first routine to be used when the vector is called.

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

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

Note that this SWI cannot be re-entered as it disables IRQs.

Examples

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

Related SWIs

SWI OS_Release (on page 10)
SWI OS_CallAVector (on page 11)
SWI OS_AddToVector (on page 12)

OS_Release (SWI &20)

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

On entry

R0 = vector number (see *List of software vectors (on page 5)*)
R1 = address of routine that is to be released from vector
R2 = value given in R2 when claimed

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 not re-entrant

Use

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

Note that this SWI cannot be re-entered as it disables IRQs.

Examples

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

Related SWIs

SWI OS_Claim (on page 8)
SWI OS_CallAVector (on page 11)
SWI OS_AddToVector (on page 12)

OS_CallAVector (SWI &34)

Calls a vector directly

On entry

R0 - R8 = vector routine parameters
R9 = vector number (see *List of software vectors (on page 5)*)

On exit

R0 - R9 = depends on vector called
C flag flag pass to vector
V flag flag pass to vector

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

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

This is used for calling vectored routines which don't have any other entry point, such as some calls to *RemV (on page 23)* or *CnpV (on page 29)*. It is also used by system extensions such as the *Draw module (on page 0)*, *ColourTrans (on page 0)* and *Econet (on page 0)* modules to call their corresponding vectors.

You must not use this SWI to call *ByteV (on page 0)* and other such vectors, as the vector handlers expect entry conditions you may not provide.

Note that although this SWI is re-entrant, the vectors that it calls may not be.

Related SWIs

SWI OS_Claim (on page 8)
SWI OS_Release (on page 10)
SWI OS_AddToVector (on page 12)

OS_AddToVector (SWI &47)

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

On entry

R0 = vector number (see *List of software vectors (on page 5)*)
R1 = address of claiming routine
R2 = value to be passed in R12 when the routine is called

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 not re-entrant

Use

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

Unlike *SWI OS_Claim (on page 8)*, any earlier instances of the same routine remain on the vector chain.

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

Note that this SWI cannot be re-entered as it disables IRQs.

Related SWIs

SWI OS_Claim (on page 8)
SWI OS_Release (on page 10)
SWI OS_CallAVector (on page 11)

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 is not re-entrant

Use

When an application running in application space (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.

Note that this SWI cannot be re-entered as it disables IRQs.

Related SWIs

SWI OS_RelinkApplication (on page 14)

OS_RelinkApplication (SWI &4E)

Restore from a buffer any vectors that an application is using

On entry

R0 = pointer to buffer

On exit

R0 preserved

Interrupts

Interrupts are 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 *SWI OS_DelinkApplication (on page 13)*.

Related SWIs

SWI OS_DelinkApplication (on page 13)

Software vectors

Vector UserV (Vector &00)

Reserved vector

On entry

None

On exit

None

Interrupts

Interrupts are undefined
Fast interrupts are undefined

Processor mode

Processor is in undefined mode

Re-entrancy

Not defined

Use

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

Related APIs

None

Vector IrqV (Vector &02)

Called when an unknown IRQ is detected

On entry

None

On exit

None

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in irq mode

Re-entrancy

Vector is not re-entrant

Use

This vector is called when an unknown IRQ is detected.

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

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

Routines that claim this vector must not corrupt any registers. You must not call this vector using *SWI OS_CallAVector (on page 11)*.

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

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

See the chapter entitled *Interrupts and handling them (on page 0)* for details of how to add interrupt generating devices to your computer, and the chapter entitled *Handlers (on page 0)* for more about handlers.

Related APIs

None

Vector InsV (Vector &14)

Called to place a byte or block in a buffer

On entry

R1 = operation flag:

Bit(s)	Meaning
0-30	Buffer number
31	Clear: *Insert a byte in a buffer (on page 19)*
	Set: *Insert a block in a buffer (on page 21)*

On exit

R1 preserved
C flag flag = 1 implies insertion failed

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Re-entrancy

Vector is not re-entrant

Use

This vector is called by *SWI OS_Byte 138 (on page 0)* and *SWI OS_Byte 153 (on page 0)*. The default action is to call the ROM routine to insert byte(s) into a buffer from the system buffers.

It may also be called using *SWI OS_CallAVector (on page 11)*. It must be called with interrupts disabled (the OS_Bytes do this automatically), therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.

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

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

To use different sized system buffers under RISC OS 2, you must provide handlers for all of *InsV (on page 17)*, *RemV (on page 23)* and *CnpV (on page 29)*. Under later versions of RISC OS you must instead use the buffer manager SWIs *SWI Buffer_Create (on page 0)* or *SWI Buffer_Register (on page 0)*.

See also the chapter entitled *Buffers (on page 0)*, and the chapter entitled *Buffer manager (on*

page 0).

Related SWIs

SWI OS_Byte 138 (on page 0)
SWI OS_Byte 153 (on page 0)
SWI OS_CallAVector (on page 11)
SWI Buffer_Create (on page 0)
SWI Buffer_Register (on page 0)

Vector InsV 0 InsertByteInBuffer (Vector &14)

Insert a byte in a buffer

On entry

R0 = byte to be inserted
R1 = operation flag:

Bit(s)	Meaning
0-30	Buffer number
31	Clear: Byte insertion

On exit

R0 preserved
R1 preserved
R2 corrupted
C flag flag:

Value	Meaning
1	Insertion failed

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Re-entrancy

Vector is not re-entrant

Use

This vector is called by *SWI OS_Byte 138 (on page 0)* and *SWI OS_Byte 153 (on page 0)*. The default action is to call the ROM routine to insert byte(s) into a buffer from the system buffers.

It may also be called using *SWI OS_CallAVector (on page 11)*. It must be called with interrupts disabled (the OS_Bytes do this automatically), therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.

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

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

To use different sized system buffers under RISC OS 2, you must provide handlers for all of *InsV*

(on page 17), *RemV (on page 23)* and *CnpV (on page 29)*. Under later versions of RISC OS you must instead use the buffer manager SWIs *SWI Buffer_Create (on page 0)* or *SWI Buffer_Register (on page 0)*.

See also the chapter entitled *Buffers (on page 0)*, and the chapter entitled *Buffer manager (on page 0)*.

Related SWIs

SWI OS_Byte 138 (on page 0)
SWI OS_Byte 153 (on page 0)
SWI OS_CallAVector (on page 11)
SWI Buffer_Create (on page 0)
SWI Buffer_Register (on page 0)

Related vectors

InsV (on page 17)

Vector InsV 1 InsertBlockInBuffer (Vector &14)

Insert a block in a buffer

On entry

R1 = operation flag:

Bit(s)	Meaning
0-30	Buffer number
31	Set: Block insertion

R2 = pointer to first byte of data to be inserted
R3 = number of bytes to insert

On exit

R0 preserved
R1 preserved
R2 = pointer to remaining data to be inserted
R3 = number of bytes still to be inserted
C flag flag:

Value	Meaning
1	Insertion failed

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Re-entrancy

Vector is not re-entrant

Use

This vector is called by *SWI OS_Byte 138 (on page 0)* and *SWI OS_Byte 153 (on page 0)*. The default action is to call the ROM routine to insert byte(s) into a buffer from the system buffers.

It may also be called using *SWI OS_CallAVector (on page 11)*. It must be called with interrupts disabled (the OS_Bytes do this automatically), therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.

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

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

To use different sized system buffers under RISC OS 2, you must provide handlers for all of *InsV (on page 17)*, *RemV (on page 23)* and *CnpV (on page 29)*. Under later versions of RISC OS you must instead use the buffer manager SWIs *SWI Buffer_Create (on page 0)* or *SWI Buffer_Register (on page 0)*.

See also the chapter entitled *Buffers (on page 0)*, and the chapter entitled *Buffer manager (on page 0)*.

Related SWIs

SWI OS_Byte 138 (on page 0)
SWI OS_Byte 153 (on page 0)
SWI OS_CallAVector (on page 11)
SWI Buffer_Create (on page 0)
SWI Buffer_Register (on page 0)

Related vectors

InsV (on page 17)

Vector RemV (Vector &15)

Called to remove a byte or block from a buffer

On entry

R1 = operation flag:

Bit(s)	Meaning
0-30	Buffer number
31	Clear: *Remove a byte from a buffer (on page 25)*
	Set: *Remove a block from a buffer (on page 27)*

V flag flag:

Value	Meaning
0	Data should be removed
1	Buffer to be examined only

On exit

R1 preserved

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Re-entrancy

Vector is not re-entrant

Use

This vector is called by *SWI OS_Byte 145 (on page 0)* and *SWI OS_Byte 152 (on page 0)*. The default action is to call the ROM routine to examine or remove byte(s) from the system buffers.

It may also be called using *SWI OS_CallAVector (on page 11)*. It must be called with interrupts disabled (the OS_Bytes do this automatically), therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.

The C flag is used to indicate if the operation failed; if C=1 then it was not possible to remove/examine all the specified data, or the specified byte.

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

To use different sized system buffers under RISC OS 2, you must provide handlers for all of InsV, RemV and CnpV. Under later versions of RISC OS you must instead use the buffer manager SWIs *SWI Buffer_Create (on page 0)* or *SWI Buffer_Register (on page 0)*.

See also the chapter entitled *Buffers (on page 0)*, and the chapter entitled *Buffer manager (on page 0)*

Related SWIs

SWI OS_Byte 145 (on page 0)
SWI OS_Byte 152 (on page 0)
SWI OS_CallAVector (on page 11)
SWI Buffer_Create (on page 0)
SWI Buffer_Register (on page 0)

Vector RemV 0 RemoveByteFromBuffer (Vector &15)

Remove a byte from a buffer

On entry

R1 = operation flag:

Bit(s)	Meaning
0-30	Buffer number
31	Clear: Byte removal

V flag flag:

Value	Meaning
0	Data should be removed
1	Buffer to be examined only

On exit

R0 = next byte to be removed (examine option), or corrupted (remove option)
R1 preserved
R2 = byte removed (remove option), or corrupted (examine option)
C flag flag:

Value	Meaning
1	Buffer was empty on entry

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Re-entrancy

Vector is not re-entrant

Use

This vector is called by *SWI OS_Byte 145 (on page 0)* and *SWI OS_Byte 152 (on page 0)*. The default action is to call the ROM routine to examine or remove byte(s) from the system buffers.

It may also be called using *SWI OS_CallAVector (on page 11)*. It must be called with interrupts disabled (the OS_Bytes do this automatically), therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.

The C flag is used to indicate if the operation failed; if C=1 then it was not possible to remove/examine all the specified data, or the specified byte.

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

To use different sized system buffers under RISC OS 2, you must provide handlers for all of InsV, RemV and CnpV. Under later versions of RISC OS you must instead use the buffer manager SWIs *SWI Buffer_Create (on page 0)* or *SWI Buffer_Register (on page 0)*.

See also the chapter entitled *Buffers (on page 0)*, and the chapter entitled *Buffer manager (on page 0)*

Related SWIs

SWI OS_Byte 145 (on page 0)
SWI OS_Byte 152 (on page 0)
SWI OS_CallAVector (on page 11)
SWI Buffer_Create (on page 0)
SWI Buffer_Register (on page 0)

Related vectors

RemV (on page 23)

Vector RemV 1 RemoveBlockFromBuffer (Vector &15)

Remove a block from a buffer

On entry

R1 = operation flag:

Bit(s)	Meaning
0-30	Buffer number
31	Set: Block removal

R2 = pointer to block to be filled
R3 = number of bytes to place into block
V flag flag:

Value	Meaning
0	Data should be removed
1	Buffer to be examined only

On exit

R0 preserved
R1 preserved
R2 = pointer to updated buffer position
R3 = number of bytes still to be removed
C flag flag:

Value	Meaning
1	Buffer was empty on entry

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Re-entrancy

Vector is not re-entrant

Use

This vector is called by *SWI OS_Byte 145 (on page 0)* and *SWI OS_Byte 152 (on page 0)*. The default action is to call the ROM routine to examine or remove byte(s) from the system buffers.

It may also be called using *SWI OS_CallAVector (on page 11)*. It must be called with interrupts disabled (the OS_Bytes do this automatically), therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.

The C flag is used to indicate if the operation failed; if C=1 then it was not possible to remove/examine all the specified data, or the specified byte.

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

To use different sized system buffers under RISC OS 2, you must provide handlers for all of InsV, RemV and CnpV. Under later versions of RISC OS you must instead use the buffer manager SWIs *SWI Buffer_Create (on page 0)* or *SWI Buffer_Register (on page 0)*.

See also the chapter entitled *Buffers (on page 0)*, and the chapter entitled *Buffer manager (on page 0)*

Related SWIs

SWI OS_Byte 145 (on page 0)
SWI OS_Byte 152 (on page 0)
SWI OS_CallAVector (on page 11)
SWI Buffer_Create (on page 0)
SWI Buffer_Register (on page 0)

Related vectors

RemV (on page 23)

Vector CnpV (Vector &16)

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

On entry

R1 = buffer number
The V flag and C flag encode the operation required

On exit

R0 corrupted
R1 = count (LSB):

Bit(s)	Meaning
0-7	Least significant 8 bits of count, if V flag = 0 on entry; else preserved

R2 = count (MSB):

Bit(s)	Meaning
0-23	Most significant 24 bits of count, if V flag = 0 on entry; else preserved

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Re-entrancy

Vector is not re-entrant

Use

This vector is called by *SWI OS_Byte 15 (on page 0)*, *SWI OS_Byte 21 (on page 0)* and *SWI OS_Byte 128 (on page 0)*. The default action is to call the ROM routine to count the number of entries in a buffer, or to flush the contents of a buffer.

It may also be called using *SWI OS_CallAVector (on page 11)*. It must be called with interrupts disabled (the OS_Bytes do this automatically), therefore code on the vector can only be entered with interrupts disabled and is not re-entrant.

This vector can be entered in either IRQ or SVC mode.

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

Value	Meaning
0	count the entries in a buffer
1	flush the buffer

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

Value	Meaning
0	return the number of entries in the buffer
1	return the amount of space left in the buffer

This call also copes with buffer manager buffers.

To use different sized system buffers under RISC OS 2, you must provide handlers for all of InsV, RemV and CnpV. Under later versions of RISC OS you must instead use the buffer manager SWIs *SWI Buffer_Create (on page 0)* or *SWI Buffer_Register (on page 0)*.

See also the chapter entitled *Buffers (on page 0)*, and the chapter entitled *Buffer manager (on page 0)*

Related SWIs

SWI OS_Byte 15 (on page 0)
SWI OS_Byte 21 (on page 0)
SWI OS_Byte 128 (on page 0)
SWI OS_CallAVector (on page 11)
SWI Buffer_Create (on page 0)
SWI Buffer_Register (on page 0)

Vector UKSWIV (Vector &18)

Called when an unknown SWI instruction is issued

On entry

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

On exit

None

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

The default action is to call the Unused SWI handler, which by default returns a 'No such SWI' error. See the section entitled *Unused SWI (on page 0)* for full details.

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

See also the chapter entitled *An introduction to SWIs (on page 0)*; and the chapter entitled *Handlers (on page 0)* for more about handlers.

Related SWIs

SWI OS_UnusedSWI (on page 0)

Vector TickerV (Vector &1C)

Called every centisecond

On entry

None

On exit

None

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in IRQ or SVC mode

Re-entrancy

Not defined

Use

This vector is called every centisecond. It must never be intercepted, as this would prevent other clients from being called.

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

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

Related APIs

None

Vector DrawV (Vector &20)

Used to indirect all SWI calls made to the Draw module

On entry

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

Index	Decoded as SWI Call
0	*SWI Draw_ProcessPath (on page 0)*
2	*SWI Draw_Fill (on page 0)*
4	*SWI Draw_Stroke (on page 0)*
6	*SWI Draw_StrokePath (on page 0)*
8	*SWI Draw_FlattenPath (on page 0)*
10	*SWI Draw_TransformPath (on page 0)*

On exit

R0 - R10 = depends on SWI issued

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

See also the chapter entitled *Draw module (on page 0)*

Related SWIs

SWI Draw_ProcessPath (on page 0)
SWI Draw_Fill (on page 0)
SWI Draw_Stroke (on page 0)
SWI Draw_StrokePath (on page 0)
SWI Draw_FlattenPath (on page 0)
SWI Draw_TransformPath (on page 0)

Vector EconetV (Vector &21)

Called whenever there is activity on the Econet

On entry

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

On exit

R0 preserved
R1 preserved

Interrupts

Interrupts are undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

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

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

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

Reason code		Activity
&10	R1 = total size of data	NetFS_StartLoad
&11	R1 = amount of data transferred	NetFS_PartLoad
&12		NetFS_FinishLoad
&20	R1 = total size of data	NetFS_StartSave
&21	R1 = amount of data transferred	NetFS_PartSave
&22		NetFS_FinishSave
&30	R1 = total size of data	NetFS_StartCreate
&31	R1 = amount of data transferred	NetFS_PartCreate
&32		NetFS_FinishCreate
&40	R1 = total size of data	NetFS_StartGetBytes
&41	R1 = amount of data transferred	NetFS_PartGetBytes
&42		NetFS_FinishGetBytes
&50	R1 = total size of data	NetFS_StartPutBytes
&51	R1 = amount of data transferred	NetFS_PartPutBytes
&52		NetFS_FinishPutBytes
&60	start of a Broadcast_Wait	NetFS_StartWait
&62	end of a Broadcast_Wait	NetFS_FinishWait
&70	R1 = total size of data	NetFS_StartBroadcastLoad
&71	R1 = amount of data transferred	NetFS_PartBroadcastLoad
&72		NetFS_FinishBroadcastLoad
&80	R1 = total size of data	NetFS_StartBroadcastSave>
&81	R1 = amount of data transferred	NetFS_PartBroadcastSave
&82		NetFS_FinishBroadcastSave
&C0	start to wait for a transmission to end	Econet_StartTransmission
&C2	DoTransmit returns	Econet_FinishTransmission
&D0	start to wait for a reception to end	Econet_StartReception
&D2	WaitForReception returns	Econet_FinishReception

This vector is normally claimed by the NetStatus module, which uses the Hourglass module to display an hourglass while the Econet is busy. It passes on the call. If the Hourglass module is disabled, the default action is to do nothing. See the chapter entitled *Hourglass (on page 0)*, and the chapter entitled *NetStatus (on page 0)*.

See also the chapter entitled *NetFS (on page 0)*, the chapter entitled *NetPrint (on page 0)*, and the chapter entitled *Econet (on page 0)*.

Related APIs

None

Examples

An example program

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

The program claims *WrchV (on page 0)*, adding a routine that:

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

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

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


; do our preprocessing; as a trivial example, convert to the current case
LDRB r14, [r12]                    ; pick up upper/lowercase flag
CMP r14, #0                        ; decide which territory manager table to use
LDREQ r1, lowercase_table%
LDRNE r1, uppercase_table%
LDRB r0, [r1, r0]                  ; look up character and put back in r0


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

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


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


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


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


LDRB r14, [r12]
```

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


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


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


; could pass the call on instead by omitting r14 from the addresses
; to pull - ie use LDMVCFD r13!, {r0, r10-r12, r15}
; there was an error; set up the correct error pointer, flags, and
; claim the vector.


STR r0, [r13]                        ; save the error pointer
LDMFD r13!, {r0, r10-r12, r14, r15} ; return with V still set, and claim the vector


; reserve space to store the addresses of the territory manager case tables
.lowercase_table%
EQUD 0
.uppercase_table%
EQUD 0
]
NEXT


REM Get addresses of the territory manager case tables
SYS "Territory_LowerCaseTable",-1 TO !lowercase_table%
SYS "Territory_UpperCaseTable",-1 TO !uppercase_table%
DIM flag% 1
?flag%=0
WrchV%=3
ON ERROR SYS "XOS_Release", WrchV%, vectorcode%, flag%: PRINTREPORT$: END
SYS "OS_Claim", WrchV%, vectorcode%, flag%
REPEAT
  INPUT command$
  OSCLI command$
UNTIL command$=""
SYS "XOS_Release", WrchV%, vectorcode%, flag%
END
```

Document information

Maintainer(s): RISCOS Ltd <developer@riscos.com>

History:

Revision	Date	Author	Changes
1		ROL	Initial version
2	04 Mar 2004	ROL	Filled out from original document • Summary of vectors added. • IrqV documented. • ColourV linked to (in ColourTrans documentation). • PaletteV documented.