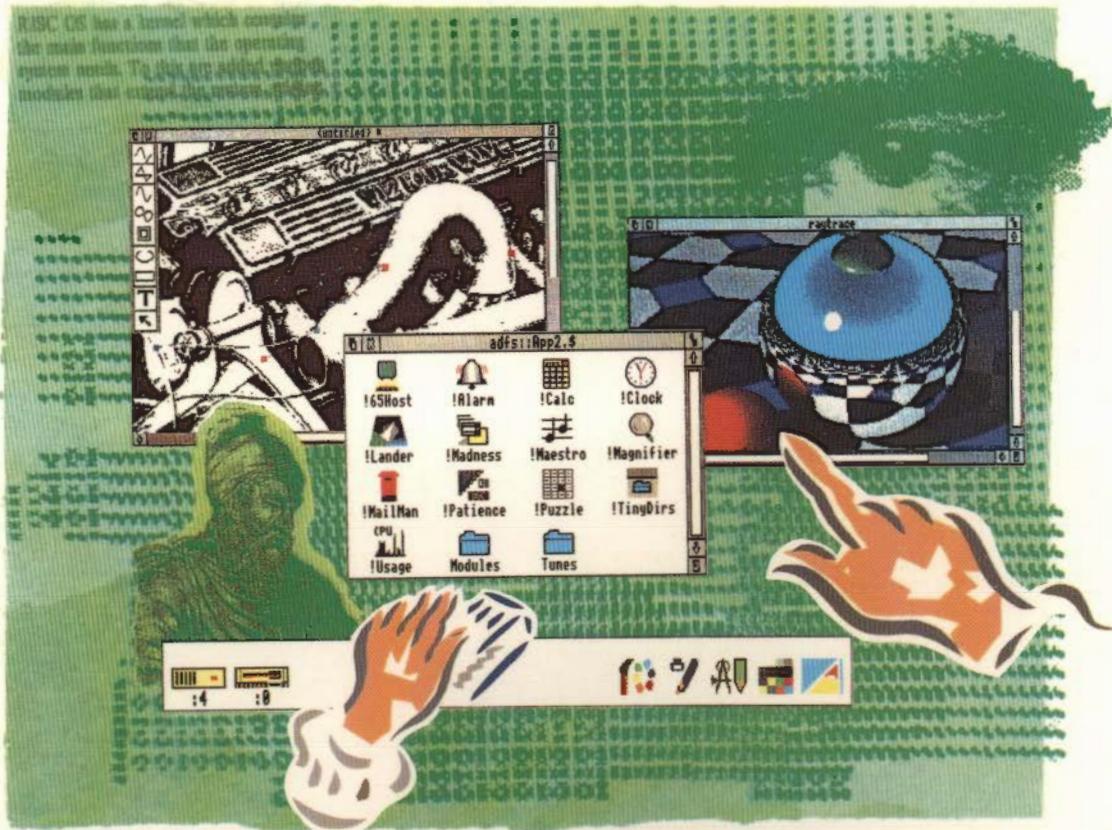


RISC OS PROGRAMMER'S REFERENCE MANUAL Volume IV

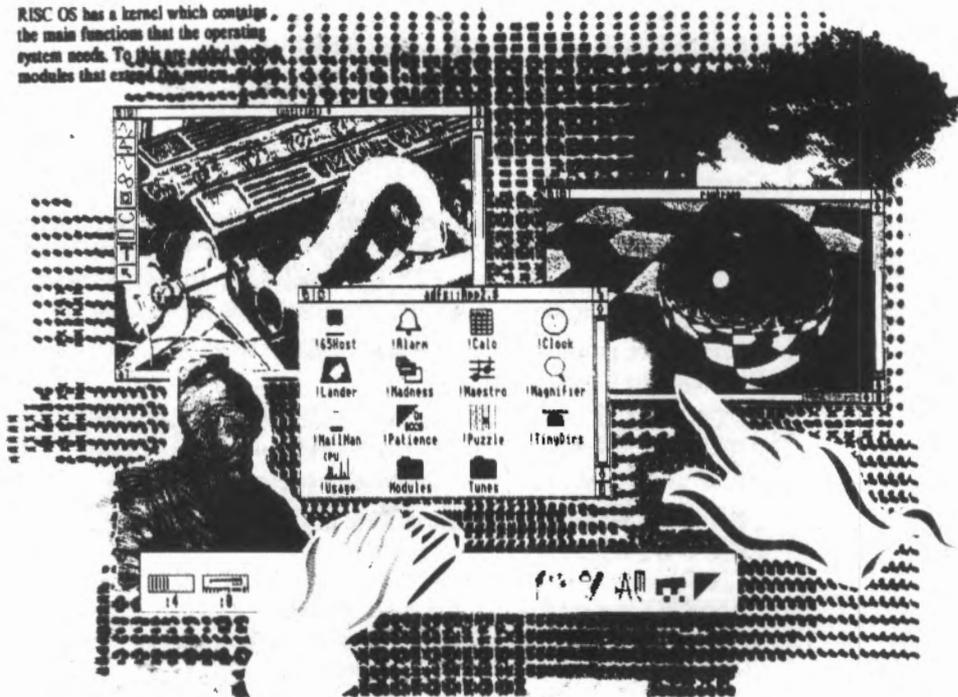


RISC OS

PROGRAMMER'S REFERENCE MANUAL

Volume IV

RISC OS has a kernel which contains the main functions that the operating system needs. To this are added other modules that extend the system.



Copyright © Acorn Computers Limited 1989

Neither the whole nor any part of the information contained in, or the product described in this manual may be adapted or reproduced in any material form except with the prior written approval of Acorn Computers Limited.

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'.

ACORN, ACORNSOFT, ACORN DESKTOP PUBLISHER, ARCHIMEDES, ARM, ARTHUR, ECONET, MASTER, MASTER COMPACT, THE TUBE, VIEW and VIEWSHEET are trademarks of Acorn Computers Limited.

DBASE is a trademark of Ashton Tate Ltd
EPSON is a trademark of Epson Corporation
ETHERNET is a trademark of Xerox Corporation
LASERJET is a trademark of Hewlett-Packard Company
LASERWRITER is a trademark of Apple Computer Inc
LOTUS 123 is a trademark of The Lotus Corporation
MULTISYNC is a trademark of NEC Limited
POSTSCRIPT is a trademark of Adobe Systems Inc
SUPERCALC is a trademark of Computer Associates
UNIX is a trademark of AT&T
1ST WORD PLUS is a trademark of GST Holdings Ltd

Edition 1
Published 1989: Issue 1
ISBN 1 85250 063 8
Published by Acorn Computers Limited
Part number 0483,023

Contents

About this manual

Part 1: Introduction

An introduction to RISC OS	3
ARM Hardware	7
An introduction to SWIs	21
* Commands and the CLI	31
Generating and handling errors	37
OS_Byte	43
OS_Word	51
Software vectors	55
Hardware vectors	85
Interrupts and handling them	91
Events	113
Buffers	125
Communications within RISC OS	135

Part 2: The kernel

Character output	149
VDU drivers	207
Sprites	379
Character input	461
Time and date	549
Conversions	579
The CLI	613
Modules	621
Program Environment	729
Memory Management	773
The rest of the kernel	815

Part 3: Filing systems	FileSwitch	831	
	FileCore	1007	
	ADFS	1051	
	RamFS	1067	
	NetFS	1075	
	NetPrint	1105	
	DeskFS	1117	
	System devices	1119	
Part 4: The Window manager	The Window Manager	1125	
Part 5: System extensions	Econet	1333	
	Hourglass	1389	
	NetStatus	1397	
	ColourTrans	1399	
	The Font Manager	1425	
	Draw module	1487	
	Printer Drivers	1513	
	The Sound system	1571	
	WaveSynth	1633	
	Expansion Cards	1635	
	International module	1665	
	Debugger	1679	
	Floating point emulator	1695	
	ShellCLI	1709	
Command scripts	1713		
			In this volume
Appendices	ARM assembler	1723	
	Linker	1743	
	Procedure Call standard	1749	
	ARM Object Format	1771	
	File formats	1787	
Tables	VDU codes	1815	
	Modes	1817	
	File types	1819	
	Character sets	1823	

Part 5 - System extensions

Econet

Introduction

The Econet module provides the software needed to use Acorn's own Econet networking system. The software allows you to send and receive data over the network.

It is used by RISC OS modules such as NetFS and NetPrint, which provide network filing and printing facilities respectively. It is also used by various other Acorn products that use Econet, such as FileStores, Econet bridges, and so on.

Note that to use the Econet you must have an Econet expansion module fitted to your RISC OS computer. If you do not already have one, they are available from your Acorn supplier.

Overview

Econet is Acorn's own networking system, and the Econet module provides the necessary software to use it.

The main purpose of any networking system is to transfer data from one machine to another. Econet breaks up the data it sends into small parts which are sent using a well defined protocol.

Econet does not use buffers in the same way as most other input and output facilities that RISC OS provides. Instead the data is moved directly between the Econet hardware and memory. This means that each time data is transmitted or received, there has to be a block of memory available for the Econet software to use immediately, either to read data from or place data in.

These blocks of memory are administered by the Econet software, which uses control blocks to do so. Many of the SWIs interact with these control blocks, so you can set them up, read the status of an Econet transmission or reception, and release the control blocks memory when you have finished using them.

In the same way as files under the filing system use file handles, these control blocks also use handles. Just like file handles, your software must keep a record of them while you need to use them.

The Econet also provides a range of immediate operations, which allow you to exercise some control over the hardware of remote machines, assuming you get their co-operation. Some of these will work across the entire range of Acorn computers, whereas others are more hardware-dependant and so may only be possible on RISC OS machines.

Technical Details

Packets and frames

A single transmission of data on an Econet is called a *packet*. Packets travel across the network from the transmitting station to the receiving station. The most common form of packet is called a 'four way handshake'. A 'four way handshake' consists of *four* frames. Each of these four frames starts with the following four bytes:

- the station number of the destination station
- the network number of the destination station
- the station number of the source station
- the network number of the source station.

These four bytes are sent in this order to facilitate decoding by the software in the receiving station.

The first frame is sent by the transmitting station, it contains the usual first four bytes, the port byte (described later), and the flag byte (also described later). This first frame is called the *scout*. The receiving station then replies with the *scout acknowledge*, which consists of just the usual first four bytes. The third frame is the *data* frame; this frame has the usual first four bytes, followed by all the data to be transferred. Lastly there is a *final acknowledge* frame which is identical to the scout acknowledge frame.

This exchange of frames can be seen with the NetMonitor and is displayed something like this.

```
FE0012008099 1200FE00 FE00120048454C500D 1200FE00
```

- the transmitting station is &12 (18 in decimal)
- the receiving station is &FE (254 in decimal)
- both stations are on network zero
- the flag byte is &80
- the port byte is &99
- the data that is transmitted is &48, &45, &4C, &50, &0D.

Receiving data and using RxCB's

Successful transmission of data requires co-operation from the receiving station. A station shows that it is ready to receive by setting up a *receive control block* (or RxCB). All RxCBs are kept by the Econet software and don't need to concern you. To create an RxCB all you need to do is call a single SWI (Econet_CreateReceive – SWI &40000), telling the Econet software all the required information. The Econet software will return to you a handle which you then use to refer to this particular RxCB in any further dealings with the Econet software.

The information required by the Econet software is:

- which station(s) to accept data from
- which port number(s) to accept data on
- where to put the data when it arrives.

It is important you note that when the data arrives from the transmitting station it is not buffered at all – it is taken directly from the hardware and placed in memory at the address you specify. This area of memory is referred to as a buffer (in this case a *receive buffer*). A consequence of this is that memory used for receiving Econet packets must be available at all times whilst the relevant RxCB is open. You must not use memory in application space if your program is to run within the Desktop environment.

The Econet software keeps a list of all the open RxCBs. When a packet comes in it is checked to see if it matches any of the currently open RxCBs:

- if it doesn't then the receiving software indicates this to the transmitting software by not sending a scout acknowledge frame
- if it does then the receiving software sends out a scout acknowledge, and then copies the data frame into the corresponding buffer
- if the data frame overruns the buffer then the receiving software does not send the final acknowledge frame.

Status of RxCB's

All RxCBs have a status value. These values are tabulated below.

- 7 Status_RxReady
- 8 Status_Receiving
- 9 Status_Received

The status of a particular RxCB can be read using the `Econet_ExamineReceive` call (SWI &40001); this takes the receive handle of an RxCB and returns its status.

When an RxCB has been received into, its status will change from `RxReady` to `Received`; usually, you will then call `Econet_ReadReceive` (SWI &40002). This returns information about the reception; most importantly it tells you how much data was received – which can be anything from zero to the size of the buffer. It also returns the value of the flag byte.

The port, station, and network are also returned; these are useful because you can open an RxCB that allows reception on any port or from any station.

Abandoning RxCB's

It is very important that when RxCBs are no longer required, either because they have been received into, or because they have not been received into within a certain time, that they are removed from the system. You do so by calling the SWI `Econet_AbandonReceive` (SWI &40003). The major function of this call is to return to the RMA the memory that the Econet software used to hold the RxCB; obviously if RxCBs are not abandoned, they will consume memory which will not automatically be recovered by the system.

Receiving data using a single SWI

The usual sequence of operations required for software to receive data is as follows: First call SWI `Econet_CreateReceive`, then make numerous calls to SWI `Econet_ExamineReceive` until either a reception occurs, a time out occurs, or the user interferes (by pressing *Escape* for instance). Then read the RxCB if it has been received into. Finally, abandon the RxCB.

To make this task easier the Econet software provides a single SWI (`Econet_WaitForReception` – SWI &40004) which does the polling, the reading, and the abandoning for you. To call SWI `Econet_WaitForReception` you must pass in:

- the receive handle
- the amount of time you are prepared to wait
- a flag which indicates whether you wish the call to return if the user presses the *Escape* key.

Econet_WaitForReception returns one of four status values:

- 8 Status_Receiving
- 9 Status_Received
- 10 Status_NoReply
- 11 Status_Escape

The call will return as soon as a reception occurs; when this happens the status is *Received*. If the time limit expires then the status is usually *NoReply*, but if reception had started just after the timeout, and so was then abandoned, the status will be *Receiving*. This is not a very likely case. If the escapable flag is set then pressing the Escape key causes the call to return with the *Escape* status.

Transmitting data and using TxCB's

Transmission is roughly similar to reception; a single SWI (Econet_StartTransmit – SWI &40006) is all that is required to get things started. This call requires the following information:

- the destination station (and network)
- the port number to transmit on
- the flag byte to send
- the address and length of the data to send.

SWI Econet_StartTrasmit returns a handle. These handles are distinct from the handles used by the receive SWIs.

There is a limit of 8 kbytes on the size of data you can send with this call.

Status of TxCB's

To check the progress of your transmission you can call Econet_PollTransmit (SWI &40007). This returns the status of the particular TxCB, which will be one of seven possible values:

- 0 Status_Transmitted
- 1 Status_LineJammed
- 2 Status_NetError
- 3 Status_NotListening
- 4 Status_NoClock
- 5 Status_TxReady
- 6 Status_Transmitting

Status_Transmitted means that your transmission has completed OK and that the data has been received by the destination machine. *Status_TxReady* means that your transmission is waiting to start, either because the Econet is busy receiving or transmitting something else, or your transmission is queued (see later for more details of this). *Status_Transmitting* is obvious; so too is *Status_NoClock*, which means that the Econet is not being clocked, or more likely your station is not plugged into the Econet. *Status_LineJammed* means that the Econet software was unable to gain access to the Econet; this may be because other stations were transmitting, but it is more likely that there is a fault in the Econet cabling somewhere. *Status_NotListening* is returned when the destination station doesn't send back a scout acknowledge frame; this is usually because the destination station doesn't have a suitable open receive block. *Status_NetError* will be returned if some part of the four way handshake is missing or damaged; the usual cause of this status is the sender sending more data than the receiver has buffer space for, so the receiver doesn't send back the final acknowledge frame.

Retrying transmissions

Status returns like *NotListening* and *NetError* can also be caused by transient problems with the Econet such as electrical noise, or by the receiving station using its floppy disc. Because of this it is usual to try more than once to send a packet if these status returns occur. To make this easier for you the Econet software can automatically perform these extra attempts for you. These retries are controlled by passing two further values in to the `Econet_StartTransmit SWI`:

- the number of times to try, referred to as the Count
- the amount of time to wait between tries, referred to as the Delay.

If the Count is either zero or one then only one attempt to transmit will take place. If the Count is two or more then retries will occur, at the specified interval (given in centi-seconds). To give an example as it would be written in BASIC V:

```
10 DIM Buf% 20
20 Port%=99: Station%=7: Network%=0
50 SYS "Econet_StartTransmit",0,Port%,Station%,Network%,Buf%,20,3,100 TO Tx%
60 END
```

When this partial program was RUN it would try to transmit immediately, probably before the program reached the END statement. If this transmission failed with either *Status_NotListening* or *Status_NetError*, then the Econet

software would wait for one second (100 centi-seconds) and try again. If this also failed then the software would wait a further second and try for a third time. The status of the final (in this case third) transmission would be the status finally stored in the TxCB; this could be read using SWI Econet_PollTransmit. To see this we could add some extra lines to the example program.

```
30 TxReady%=5
40 Transmitting%=6
60 REPEAT
70   SYS "Econet_PollTransmit", Tx% TO Status%
80   PRINT Status%
90 UNTIL NOT ((Status%=TxReady%) OR (Status%=Transmitting%))
100 END
```

Now the program will show us the status of the TxCB. We would be very unlikely to see the status value ever be Status_Transmitting since it will only have this value for about 90µs during the two seconds it is retrying for. But it is most important that your software should be able to handle such a situation without error.

Abandoning TxCB's

As with receptions it is most important that memory used for transmitting Econet packets **must** be available at all times whilst the relevant TxCB is open. You **must not** use memory in application space if your program is to run within the Desktop environment. This is because like receptions, transmissions move data directly from memory at the address you specify to the hardware. Also, as with receptions, it is important to inform the Econet software that you have finished with your transmission and that memory required for the internal TxCB may be returned to the RMA. You do this by calling Econet_AbandonTransmit (SWI &40008) with the appropriate TxHandle.

```
100 SYS "Econet_AbandonTransmit", Tx% TO FinalStatus%
110 PRINT "The final status was ";FinalStatus%
```

Transmitting data using a single SWI

To make this start, poll, and abandon sequence easier for you the Econet software provides it all as a single call (Econet_DoTransmit – SWI &40009)). This call has the same inputs as SWI Econet_StartTransmit, but instead of returning a handle it returns the final status. Using this call our program would look like this:

Converting a status to an error

```
10 DIM Buf% 20
20 Port%=99: Station%=7: Network%=0
40 SYS"Econet_DoTransmit",0,Port%,Station%,Network%,Buf%,20,3,100 TO Status%
50 PRINT "The final status was ";Status%
```

As you can see this makes things a lot easier. As an aid to presenting these status values to the user there are two SWI calls to convert status values to a textual form, the most frequently used of which is the call Econet_ConvertStatusToError (SWI &4000C). This call takes the status and returns an error with the appropriate error number and an appropriate string describing the error. For instance we could add an extra line to our final program.

```
60 SYS "Econet_ConvertStatusToError", Status%
```

Copying the error to RAM

Our program will now RUN and always have an error, in this case the error "Not listening at line 50". This error block is actually in the ROM so it is not possible to add to it, but it is possible to have the call to Econet_ConvertStatusToError copy the error into RAM by specifying in the call where this memory is, and how much there is:

```
60 DIM Error% 30
80 SYS "Econet_ConvertStatusToError", Status%, Error%, 30
```

This new program will function in the same manner as the previous program except that the error block will have been copied from the Econet part of the ROM into RAM (at the address given in R1). The main reason for this is to allow the Econet software to customise the error for you.

Adding station and network numbers

If the station and network numbers are added as inputs to the call, the Econet software will add them to the output string:

```
80 SYS "Econet_ConvertStatusToError", Status%, Error%, 30, Station%, Network%
```

Now the error reported will be "Station 7 not listening at line 50". It is important to stress that this is a general purpose conversion. It will convert Status_Transmitted just as well as Status_NotListening, so usually you would test the returned status from Econet_DoTransmit, and only convert status values other than Status_Transmitted into errors:

```
30 Transmitted%=0
70 IF Status%≠Transmitted% THEN PRINT "OK": END
```

The same program fragment could be written in assembler (this example, like all others in this chapter, uses the ARM assembler rather than the assembler included with BBC BASIC V – there are subtle syntax differences):

```
Tx    MOV    r0, #0
      MOV    r1, #99
      MOV    r2, #7
      MOV    r3, #0
      LDR    r4, Buffer
      MOV    r5, #20
      MOV    r6, #3
      MOV    r7, #100
      SWI    Econet_DoTransmit
      BEQ    r0, #Status_Transmitted
      LDRNE  r1, ErrorBuffer
      MOVNE  r2, #30
      SWINE  Econet_ConvertStatusToError
      MOV    pc, lr
```

Notice here in the assembler version how the return values from `Econet_DoTransmit` fall naturally into the input values required for `Econet_ConvertStatusToError`. This code fragment is not really satisfactory since no code written as either a module or a transient command should ever call the non-X form of SWIs. If the routine `Tx` is treated as a subroutine then it should look more like this:

```
Tx    STMFD  sp!, {lr}
      MOV    r0, #0
      MOV    r1, #99
      MOV    r2, #7
      MOV    r3, #0
      ADR    r4, Buffer
      MOV    r5, #20
      MOV    r6, #3
      MOV    r7, #100
      SWI    XEconet_DoTransmit
      BVS    TxExit
      TEQ    r0, #Status_Transmitted
      ADRNE  r1, ErrorBuffer
      MOVNE  r2, #30
      SWINE  XEconet_ConvertStatusToError
TxExit LDMFD  sp!, {pc}
```

This routine returns with `V` clear if all went well; if `V` is set, then on return `R0` will contain the address of a standard error block.

Converting a status to a string

The second error conversion call is `Econet_ConvertStatusToString` (SWI &4000B), which does exactly what its name suggests. The input requirements are very similar to the string conversion SWIs supported by RISCOS. In this case you pass the status value, a buffer address, and the length of the buffer. As with `Econet_ConvertStatusToError` you can also pass the station and network numbers, which will be included in the output string. To illustrate this the assembler routine shown above is changed to print the status on the screen:

```
Tx      STMPD    sp!, {lr}
        MOV     r0, #0
        MOV     r1, #99
        MOV     r2, #7
        MOV     r3, #0
        ADR     r4, Buffer
        MOV     r5, #20
        MOV     r6, #3
        MOV     r7, #100
        SWI     XEconet_DoTransmit
        BVS     TxExit
        TEQ     r0, #Status_Transmitted
        BEQ     TxExit
        ADR     r1, TextBuffer
        MOV     r2, #50
        MOV     r5, r0                ; Save the status value
        SWI     XOS_ConvertCardinal1
        MOVVC   r0, r5                ; Recall status if no error
        SWIVC   XEconet_ConvertStatusToString
        ADRVC   r0, TextBuffer
        SWIVC   XOS_Write0
TxExit  LDMFD   sp!, {pc}
```

Flag bytes

The flag byte is sent from the transmitting station to the receiving station and can be treated as an extra seven bits of data. By convention, it is used as a simple way of distinguishing different types of packet sent to the same port, and it is worth you doing the same.

This is most useful in server type applications where it is often the case that similar data can be sent for different purposes, or some sorts of data are outside the normal scope. An example is a server that takes requests for teletext pages, but can also return the time. A different value for the flag byte allows the server to differentiate time requests from normal traffic. Another example is the printer server protocol, which uses the flag byte to indicate the packet that is the last in the print job, without having to change the data part of the packet.

Port bytes

The port byte is used in the receiving station to distinguish traffic destined for particular applications or services.

For instance the printer server protocol uses port &D1 for all its connect, data transfer, and termination traffic, whereas the file server uses port &99 for all its incoming commands. This use of separate ports for separate tasks is also exploited further by the file server protocol in that every single request for service by the user can use a different port for its reply. This prevents traffic getting confused.

The Econet software provides some support for you to use ports by providing an allocation service for port numbers. Port numbers should, if possible, be allocated for all incoming data.

Software that requires the use of fixed port numbers, like NetFS and NetPrint, can claim these fixed ports by calling `Econet_ClaimPort` (SWI &40015). This call takes a port number as its only argument. When these claimed ports are no longer required (when the module dies for instance) it can be 'returned' by calling `SWI Econet_ReleasePort` (SWI &40012).

Other software that would like a port number allocated to it can call `Econet_AllocatePort` (SWI &40013), which will return a port number. While this port number is allocated no other calls to `Econet_AllocatePort` will return that number, until it is 'returned' by calling `Econet_DeAllocatePort` (SWI &40014) with the port number as an input. The NetFS software uses this method of allocation and deallocation to get ports to use as reply ports in the file server protocol. The Econet software keeps a table in which it records the state of each port number: this can be either free, claimed or allocated.

Freeing ports

Ports that have been claimed will not be allocated, and can only be freed by calling `SWI Econet_ReleasePort`. Calling `SWI Econet_DeAllocatePort` will return an error if the port is claimed rather than allocated. Ports that have been allocated can not be claimed, and in fact an attempt to claim an allocated port will return an error. You should be careful with software that uses allocated ports to make sure that all ports are deallocated when they are no longer required, especially after an error. The claiming and releasing of ports should likewise be carefully checked.

An example of use of the port allocator

A typical example of the use of the port allocator would be a multi-player adventure game server. The server would claim one port (eg port &1F). This port number would then be the only fixed port number in the entire protocol. When a player wished to join the game she should ask for a port to be allocated in her machine and send this port, along with all the information required to enter the game, to the game server on port &1F. If the server can't be contacted or doesn't reply within the required time the port should be deallocated and an error returned. When the server receives this packet it should check the user's entry data; if this is OK it should then allocate a port for that user and return it, along with any other information required to start the game off. When the user wants to quit the game the server should deallocate its user's port, then send the last reply to the user. The user should deallocate the port when the reply arrives or if the server doesn't reply soon enough.

To illustrate this example the user entry routine is shown below; note that this routine is coded for clarity rather than size or efficiency.

```
Entry  STMFD   sp!, {r0-r8,lr} ; R0 points to the text string
      SWI     XEconet_AllocatePort
      BVS    Exit

      STRB   r0, Server_ReplyPort
      LDR    r1, Server_Station
      LDR    r2, Server_Network
      ADR    r3, Buffer
      MOV    r4, #?Buffer
      SWI    XEconet_CreateReceive
      BVS    DeAllocateExit
      MOV    r8, r0          ; Preserve the RxHandle

      LDR    r1, [ sp, #0 ] ; Address of text string to copy
      ADR    r4, Buffer      ; Get buffer to copy into
      MOV    r5, #0        ; Index into Tx Buffer
      LDRB   r0, Server_ReplyPort
      STRB   r0, [ r4, r5 ] ; Send the port for the server
CopyLoop
      ADD    r5, r5, #1
      CMP    r5, #?Buffer  ; Have we run out of buffer?
      BHS   BufferOverflow
      LDRB   r0, [ r1 ], #1 ; Pick up byte and move to next one
      CMP    r0, #" "      ; Is this a control character?
      MOVLE  r0, #CR       ; Terminate as the server expects
      STRB   r0, [ r4, r5 ]
      BGT    CopyLoop     ; Loop back for the next byte
      ADD    r5, r5, #1    ; Set entry conditions for Tx

      MOV    r0, #0
```

```

MOV     r1, #EntryPort ; A constant
LDR     r2, Server_Station
LDR     r3, Server_Network
LDR     r6, Server_TxDelay
LDR     r7, Server_TxCount
SWI     XEconet_DoTransmit
BVS     DeAllocateExit
TEQ     r0, #Status_Transmitted
BEQ     WaitForReply
ConvertEconetError
ADR     r1, Buffer ; Convert status and exit
MOV     r2, #?Buffer
SWI     XEconet_ConvertStatusToError
B       DeAllocateExit

WaitForReply
MOV     r0, r8
LDR     r1, Server_RxDelay
MOV     r2, #0 ; Don't allow ESCape
SWI     XEconet_WaitForReception
BVS     DeAllocateExit
TEQ     r0, #Status_Received
BNE     ConvertEconetError

LDR     r0, Buffer ; Get server return code
CMP     r0, #0 ; Has there been an error?
ADR     r0, Buffer ; Get address of reply
BNE     DeAllocateExit ; Yes, process error
LDRB   r1, [ r0, #4 ] ; Load server's port
STRB   r1, Server_CommandPort

Exit
STRVS  r0, [ sp, #0 ] ; Poke error into return regs
LDMFD  sp!, {r0-r8,pc} ; Return to caller

BufferOverflowError
&      ErrorNumber_BufferOverflow
=      "Command too long for buffer", 0
ALIGN

BufferOverflow
ADR     r0, BufferOverflowError
DeAllocateExit
MOV     r1, r0 ; Preserve the original error
LDRB   r0, Server_ReplyPort
SWI     XEconet_DeAllocatePort
MOV     r0, r1 ; Ignore deallocation errors
CMP     pc, #080000000 ; Set V
B       Exit ; Exit through common point

```

Points to notice in the example are:

- the careful use of a single exit point
- the consistent return of errors (no matter what type)
- the opening of the receive block before doing the transmit
- the use of the 'X' form of SWIs.

It should be noted that the routine uses and manipulates global state as well as taking specific input and returning specific output.

Econet events

To allow Econet based programs to be kinder to other applications within the machine, it is possible for your program to be 'notified' when either a reception occurs or a transmission completes. This means that other applications can be using the time that your program would have spent polling, either inside `Econet_DoTransmit` or inside `Econet_WaitForReception`. This 'notification' is carried by an event. There are separate events for reception and for completion of transmission. These two events are:

```
14 Event_Econet_Rx
15 Event_Econet_Tx
```

On entry to the event vector:

- R0 will contain the event number, either `Event_Econet_Rx` or `Event_Econet_Tx`
- R1 will contain the receive or transmit handle as appropriate
- R2 will contain the status of the completed operation.

The status for receive will always be `Status_Received`, but for transmit it will indicate how the transmission completed. These events can be enabled and disabled in the normal way using `OS_Byte` calls.

Using events from the Wimp

If your program is a client of the Wimp then all your event routine need do is set a flag that your main program polls in its main Wimp polling loop, when the event happens.

Setting up background tasks

```
Event  TEQ    r0, #Event_Econet_Rx
        TEQNE r0, #Event_Econet_Tx
        MOVNE pc, lr          ; If not, exit as fast as possible

        STMFD sp!, { lr }    ; Must preserve all regs for others
        ADR   r14, ForegroundFlag
        STR   pc, [ r14 ]    ; Set flag with non-zero value
        LDMFD sp!, { pc }    ; Return, without claiming vector
```

Since the interfaces required for reception and transmission can be called from within event routines, you can set up background tasks that make full use of the facilities offered by Econet. Note that it is important to check that the handle offered in the event belongs to your program, since there may well be many programs using this facility. The example given below is of a simple background server for sending out the time. Not all of the code needed is shown, just the event routine:

```
Start  MOV    r0, #EventV      ; The vector we want to get on is the Event
        ADR   r1, Event        ; Where to get when it happens
        MOV   r2, #0           ; Required so that we can release
        SWI   XOS_Claim

        MOVVC r0, #14          ; Enable event
        MOV   r1, #Event_Econet_Rx
        SWIVC XOS_Byte
        MOVVC r0, #14          ; Enable event
        MOV   r1, #Event_Econet_Tx
        SWIVC XOS_Byte

        MOVVC r0, #CommandPort ; First open the reception
        MOV   r1, #0           ; From any station
        MOV   r2, #0           ; From any net
        ADR   r3, Buffer
        MOV   r4, #?Buffer
        SWIVC XEconet_CreateReceive
        STRVC r0, RxHandle
        MOV   pc, lr

Event  TEQ    r0, #Event_Econet_Rx
        BNE   LookForTx
        LDR   r0, RxHandle     ; Get our global state
        TEQ   r0, r1          ; Is it for us?
        MOVNE r0, #Event_Econet_Rx
        MOVNE pc, lr          ; If not, exit as fast as possible

        STMFD sp!, { r3-r7 }  ; Only R1 and R2 are free for use
        MOV   r0, r1          ; Receive handle
        SWI   XEconet_ReadReceive ; R4.R3 is the reply address
        BVS   Exit
```

```

MOV     r6, r3           ; Save the station number for later
MOV     r0, #Module_Claim
MOV     r3, #8 + 5       ; Two words and five bytes required
SWI     XOS_Module       ; Memory MUST come from RMA
BVS     Exit

ADD     r1, r2, #8       ; Get the address of the 5 bytes
MOV     r0, #3           ; Set OS_Word reason code
STRB   r0, [ r1 ]       ; Read as a five byte time
MOV     r0, #14          ; Read from the real time clock
SWI     XOS_Word
BVS     Exit

MOV     r0, #0           ; Flag byte
MOV     r3, r4           ; Network number
MOV     r4, r1           ; Get the address of the 5 bytes
LDRB   r1, [ r5 ]       ; The reply port the client sent
MOV     r2, r6           ; Station number
MOV     r5, #5           ; Number of bytes to send
MOV     r6, #ReplyCount
MOV     r7, #ReplyDelay
SWI     XEconet_StartTransmit
BVS     Exit

SUB     r4, r2, #8       ; Note that the exit register is R2 not R4
STR     r0, [ r4, #4 ]   ; Save TxHandle in record
ADR     r1, TxList      ; Address of the head of the list
LDR     r2, [ r1, #0 ]   ; Head of the list
STR     r2, [ r4, #0 ]   ; Add the list to new record
STR     r4, [ r1, #0 ]   ; Make this record the list head

MOV     r0, #CommandPort ; Now re-open the reception
MOV     r1, #0           ; From any station
MOV     r2, #0           ; From any net
ADR     r3, Buffer
MOV     r4, #?Buffer
SWI     XEconet_CreateReceive
STRVC   r0, RxHandle

Exit
LDMFD  sp!, { r3-r7, pc } ; Return claiming vector

LookForTx
TEQ     r0, #Event_Econet_Tx
MOVNE   pc, lr
STMFDD sp!, { r3, lr } ; Get two extra registers
ADR     r3, TxList      ; The address of the head of list
LDR     r14, [ r3 ]     ; The first record in the list
B       StartLooking

NextTx
MOV     r3, r14         ; Search the next list entry
LDR     r14, [ r3 ]     ; Get the link address

StartLooking

```

```

CMP     r14, #0          ; Is this the end of the list?
MOVLE   r0, #Event_Econet_Tx ; Restore entry conditions
LDMLEFD sp!, { r3, pc } ; Return, continuing to next owner
LDR     r0, [ r14, #4 ] ; Get the handle for this record
TEQ     r0, r1          ; Is this event one of ours?
BNE     NextTx          ; No, try next record in list

LDR     r2, [ r14 ]     ; Get the remainder of the list
STR     r2, [ r3 ]     ; Remove this record from list
MOV     r2, r14        ; The record address for later
SWI     XEconet_AbandonTransmit
MOV     r0, #Module_Free
SWI     XOS_Module     ; Return memory to RMA, ignore error
LDMFD  sp!, { r3, lr, pc } ; Return, claiming vector

```

This program also illustrates some of the more advanced features of Econet. In particular; it shows the ability to specify reception control blocks that can accept messages from more than one machine, or on more than one port. Receive control blocks like this are referred to as *wild*, as in *wild card matching* used in file name look up. Specifying either the station or network number (usually both) as zero means 'match any'. The same is true of the port number, although this facility is much less useful! This wild facility does not mean that more than one packet can be received, but rather that more than one particular packet will be acceptable. Once a packet has been received, the RxCB has Status_Received and is no longer open.

It is worth noting an implementation detail here. Receive control blocks are kept by the Econet software in a list, when an incoming scout has been received the list is scanned to find the first RxCB that matches it. To ensure that things go as one would expect the Econet software that implements the SWI Econet_CreateReceive always adds wild RxCBs to the tail of the list, and normal RxCBs to the middle of the list (between the normal and the wild ones). This ensures that when packets arrive they will be checked for exact matches before wild matches, and that if there is more than one acceptable RxCB then the one used will be the one that was opened first, i.e. first in first served.

As a complement to this concept of wild receive control blocks there are broadcast transmissions. A broadcast has both its destination station and network set to &FF, it can then be received by more than one machine. To achieve this it does not use the normal four way handshake, it is in fact a single packet. On the NetMonitor it would look something like this:

Broadcast transmissions

FFFF1200809F5052494E54200100

The broadcast address at the beginning (&FF, &FF), the source station and network (&12, &00), the control byte (&80), and the port (&9F) are the same as a normal scout frame, but then the data follows, in this case eight bytes.

Although the Eiconet software within RISC OS can transmit and receive broadcast messages of up to about a thousand bytes, other machines on Eiconet can't cope with messages of more than eight bytes without getting confused; this confusion causes them to corrupt such broadcasts. These other machines include things like FileStores and bridges, so beware! It is possible to transmit and/or receive zero to eight bytes without them being corrupted, but only broadcasts of exactly eight bytes can be received by BBC or Master computers, as well as being transported from network to network by bridges.

Transmitting a broadcast is exactly the same as transmitting a normal packet, all you need to do is set the destination station and network to &FF (not -1). Broadcasts don't return the status *Status_NotListening*, since there is no way for the transmitting station to determine whether or not its broadcast was received. Broadcasts are basically designed for locating resources, i.e. to transmit your desire to know about a particular class of thing. Anything recognising the broadcast will reply, so you know what's what and where it is. NetFS uses broadcast to find file servers by name, and NetPrint uses broadcast to find printer servers. The above example contains the ASCII text 'PRINT' and is, not surprisingly, a request for all printer servers to respond.

Immediate operations

There is a second class of network operations called immediate operations. These operations don't require the explicit co-operation of the destination machine; instead the co-operation is provided by the Eiconet software in that machine. Immediate operations are similar semantically to normal transmissions but, because they have no need for a port number, have a type instead of a flag; and most also require an extra input value. They have a separate pair of SWI calls to cause them to happen: *Eiconet_StartImmediate* (SWI &40016) and *Eiconet_DoImmediate* (SWI &40017).

The call *Eiconet_StartImmediate* returns a transmit handle in exactly the same way as *Eiconet_StartTransmit* and that handle should be polled and abandoned in the same way. The call *Eiconet_DoImmediate* returns a status just as *Eiconet_DoTransmit* does.

There are nine types of immediate operations:

1	Econet_Peek	Copy memory from the destination machine
2	Econet_Poke	Copy memory to the destination machine
3	Econet_JSR	Cause JSR/BL on the destination machine
4	Econet_UserProcedureCall	Execute User remote procedure call
5	Econet_OSProcedureCall	Execute OS remote procedure call
6	Econet_Halt	Halt the destination machine
7	Econet_Continue	Continue the destination machine
8	Econet_MachinePeek	Machine peek of the destination machine
9	Econet_GetRegisters	Return registers from the destination machine

The last one, Econet_GetRegisters, can only be transmitted by or received on RISC OS based machines, whereas all the others can be transmitted or received by BBC or Master series computers. The reason for this is that Econet_GetRegisters is specific to the ARM processor.

Econet_Peek and Poke

The poke operation is very similar to a transmit, in that data is moved from the transmitting station to the receiving station. The difference is that the address at which the data is received is supplied by the transmitting station. Peek is the inverse of poke; data is moved from the receiving station into the transmitting station.

Econet_JSR, UserProcedureCall and OSProcedureCall

JSR, UserProcedureCall, and OSProcedureCall are all very similar. They send a small quantity of data, referred to as the argument buffer or arguments, to the destination machine; they then force it to execute a particular section of code. When received a JSR actually does a BL to the address given in R1, whereas UserProcedureCall and OSProcedureCall cause events to occur. These events are:

- 8 Event_Econet_UserRPC
- 16 Event_Econet_OSProc

After reception the arguments are buffered so that they may be used by the code that is called, either directly by a BL or indirectly via an event. The format of the Arguments buffer is as follows: word 0 is the length (in bytes) of the arguments, then the arguments follow this first word and may be null (ie the length may be zero).

Conditions on entry to event code

The conditions on entry to the event code are:

R0 = Event number (either Event_Econet_UserRPC or Event_Econet_OSPProc)
R1 = Address of the argument buffer
R2 = RPC number (passed in in R1 on the transmitting station)
R3 = Station that sent the RPC
R4 = Network that sent the RPC

Conditions on entry to JSR code

The conditions on entry to code that is BL'd to for a JSR are:

R1 = Address of the argument buffer
R2 = Address of the code being executed
R3 = Station that sent the JSR
R4 = Network that sent the JSR

Format of the argument buffer

The format of the argument buffer is exactly the same in all cases. If, in the case of a JSR, the call address transmitted from the remote station is -1 (&FFFFFFF) then the execution address will be the argument buffer itself; this means that relocatable ARM code can be sent as a JSR. Registers R0 to R4 can be used as they are preserved by the Econet software, and R13 can also be used as an FD stack.

The transmission of Econet_OSPProcedureCall is not intended for use by other than system software, and is only documented here for completeness. The transmission of Econet_JSR is only provided as a compatibility feature to allow interworking with BBC and Master computers.

Econet_UserProcedure calls

The Econet_UserProcedureCall is the best method for this style of communications. It does however have some restrictions. The first of these is the most important - it is executed in the destination machine as an event caused by an interrupt, and so it has all the normal restrictions applied to interrupt code. This means that code directly executed as a result of Event_Econet_UserRPC must be fast and clean, and must not call any of the normal input or output SWI routines nor call the filing system, either directly or indirectly. This is paramount if the integrity of the destination machine is to be ensured. However, you can copy away the arguments passed and signal to a foreground task (by altering a flag) that the procedure call has arrived. It is most important that you copy the arguments away, because the buffer that they are in is only valid for the duration of the event call. This means that R1 will

point to the arguments whilst you are processing the event, but afterwards the argument buffer may be overwritten. If the requirements for the processing of the call are small then it is possible to do it all within the event. An example of this is a modification of the program presented earlier that returned the time. This new program sends the time in response to a User RPC, rather than a normal packet:

```

Start  MOV    r0, #EventV      ; The vector we want to get on is the Event
      ADR    r1, Event        ; Where to get when it happens
      MOV    r2, #0          ; Required so that we can release
      SWI    XOS_Claim

      MOVVC  r0, #14          ; Enable event
      STRVC  r0, ClaimedFlag ; Set it to a non-zero value
      MOV    r1, #Event_Econet_UserRPC
      SWIVC  XOS_Byte
      MOVVC  r0, #14          ; Enable event
      MOV    r1, #Event_Econet_Tx
      SWIVC  XOS_Byte
      MOV    pc, lr

Event  TEQ    r0, #Event_Econet_UserRPC
      BNE    LookForTx
      TEQ    r2, #RPC_SendTime ; Is it for us?
      MOVNE  pc, lr          ; If not, exit as fast as possible

      LDR    r0, [ r1, #0 ] ; Get size of arguments
      TEQ    r0, #1          ; Check that it is right
      MOVNE  r0, #Event_Econet_UserRPC ; Restore exit registers
      MOVNE  pc, lr          ; If not, exit as fast as possible

      STMFD  sp!, { r5-r7 } ; Only R1 to R4 are free for use
                          ; R4.R3 is the reply address
      MOV    r6, r3          ; Save the station number for later
      MOV    r5, r1          ; Preserve arguments pointer
      MOV    r0, #Module_Claim
      MOV    r3, #8 + 5      ; Two words and five bytes required
      SWI    XOS_Module      ; Memory MUST come from RMA
      BVS   Exit

      ADD    r1, r2, #8      ; Get the address of the 5 bytes
      MOV    r0, #3          ; Set OS_Word reason code
      STRB  r0, [ r1 ]       ; Read as a five byte time
      MOV    r0, #14         ; Read from the real time clock
      SWI    XOS_Word
      BVS   Exit

      MOV    r0, #0          ; Flag byte
      MOV    r3, r4          ; Network number
      MOV    r4, r1          ; Get the address of the 5 bytes
      LDRB  r1, { r5, #4 } ; The reply port the client sent
      MOV    r2, r6          ; Station number

```

```

MOV     r5, #5           ; Number of bytes to send
MOV     r6, #ReplyCount
MOV     r7, #ReplyDelay
SWI     XEconet_StartTransmit
BVS     Exit

SUB     r4, r2, #8       ; Note that the exit register is R2 not R4
STR     r0, [ r4, #4 ]   ; Save TxHandle in record
ADR     r1, TxList       ; Address of the head of the list
LDR     r2, [ r1, #0 ]   ; Head of the list
STR     r2, [ r4, #0 ]   ; Add the list to new record
STR     r4, [ r1, #0 ]   ; Make this record the list head

Exit
LDMFD  sp!, { r5-r7, pc } ; Return claiming vector

LookForTx
TEQ     r0, #Event_Econet_Tx
MOVNE   pc, lr           ; This event has only R0 to R2
STMFD  sp!, { r3, lr }   ; Get two extra registers
ADR     r3, TxList       ; The address of the head of list
LDR     r14, [ r3 ]      ; The first record in the list
B       StartLooking,

NextTx
MOV     r3, r14          ; Search the next list entry
LDR     r14, [ r3 ]      ; Get the link address

StartLooking
CMP     r14, #0          ; Is this the end of the list?
MOVLE   r0, #Event_Econet_Tx ; Restore entry conditions
LDMLEFD sp!, { r3, pc } ; Return, continuing to next owner
LDR     r0, [ r14, #4 ]  ; Get the handle for this record
TEQ     r0, r1           ; Is this event one of ours?
BNE     NextTx          ; No, try next record in list

LDR     r2, [ r14 ]      ; Get the remainder of the list
STR     r2, [ r3 ]      ; Remove this record from list
SWI     XEconet_AbandonTransmit
MOV     r0, #Module_Free
MOV     r2, r14          ; The record address
SWI     XOS_Module       ; Return memory to RMA, ignore error
LDMFD  sp!, { r3, lr, pc } ; Return, claiming vector

```

You will notice how much simpler this program is when compared to the program shown earlier.

Econet_OSProcedure calls

There are five defined OS procedure calls for which only two have implementations under RISC OS. The five are:

- 0 Econet_OSCharacterFromNotify
- 1 Econet_OSInitialiseRemote
- 2 Econet_OSGetViewParameters
- 3 Econet_OSCauseFatalError
- 4 Econet_OSCharacterFromRemote

OSCharacterFromNotify

Econet_OSCharacterFromNotify causes the character received to be inserted in to the keyboard buffer; the code that does so looks like this:

```
InsertCharacter          ; R1 already pointing at argument buffer
MOV    r0, #138         ; Insert into buffer OS_Byte
LDRB  r2, [ r1, #4 ]   ; Get character from buffer
MOV    r1, #0          ; Buffer is keyboard
SWI    XOS_Byte
```

The NetFiler module provides a different implementation whilst the desktop is running.

OSCauseFatalError

Econet_OSCauseFatalError does exactly what its name implies. In fact it calls SWI OS_GenerateError directly from the event routine; normally this would be illegal, but since this is what the RPC is for, that is what it does. It should be observed that this can have a disastrous effect on the integrity of the machine and is not a recommended action; it is provided only for compatibility reasons.

Econet_Halt and Continue

Halt and continue are only acted upon by BBC and Master series machines; there is no implementation for receiving halt or continue on RISC OS machines or RISC iX machines.

Econet_MachinePeek

Machine peek is similar to peek, except that it is not possible to specify the address to be peeked, but rather four bytes are returned that identify the machine that is being machine peeked. Machine peek is used by some of the system software in RISC OS to quickly decide if a particular machine is present or not. The four bytes returned by machine peek are as follows:

Byte(s)	Value
1 and 2	Machine type number
3	Software version number
4	Software release number

Machine type numbers

Machine type numbers are as follows:

&0000	Reserved
&0001	Acorn BBC Micro Computer (OS 1 or OS 2)
&0002	Acorn Atom
&0003	Acorn System 3 or System 4
&0004	Acorn System 5
&0005	Acorn Master 128 (OS 3)
&0006	Acorn Electron (OS 0)
&0007	Acorn Archimedes (OS 6)
&0008	Reserved for Acorn
&0009	Acorn Communicator
&000A	Acorn Master 128 Econet Terminal
&000B	Acorn FileStore
&000C	Acorn Master 128 Compact (OS 5)
&000D	Acorn Ecolink card for Personal Computers
&000E	Acorn Unix WorkStation
&000F to &FFF9	Reserved
&FFFA	SCSI Interface
&FFFB	SJ Research IBM PC Econet interface
&FFFC	Nascom 2
&FFFD	Research Machines 480Z
&FFFE	SJ Research File Server
&FFFF	Z80 CP/M

Software version and release number

The software version and release numbers are stored in two bytes. These two bytes are encoded in packed BCD (Binary Coded Decimal) and represent a number between 0 and 99. The easiest way to display packed BCD is to print it as if it was hexadecimal data:

```
ReportStationVersion
MOV    r2, r0 ; Station number in R0
MOV    r3, r1 ; Network number in R1
MOV    r0, #Econet_MachinePeek
ADR    r4, Buffer
MOV    r5, #?Buffer
MOV    r6, #0
```

```

MOV    r7, #0
SWI    XEconet_DoImmediate
MOVVS  pc, lr
TEQ    r0, #Status_Transmitted
BEQ    PrintVersion
TEQ    r0, #Status_NotListening ; "Not listening" from Machine peek
MOVEQ  r0, #Status_NotPresent ; should return "Not present"
ADR    r1, Buffer
MOV    r2, #?Buffer
SWI    XEconet_ConvertStatusToError
MOV    pc, lr

```

```

PrintVersion
LDR    r3, [ r2 ] ; Buffer address on exit from SWI
MOV    r0, r3, ASR #24 ; Get top byte
ADR    r1, Buffer
MOV    r2, #?Buffer
SWI    XOS_ConvertHex2 ; Print BCD as hex
SWIVC  XOS_Write0 ; Display output
SWIVC  XOS_WriteI+ "." ; Divide release from version number
MOVVC  r0, r3, ASR #16 ; Get version number in place
ANDVC  r0, r0, #&FF ; Only the version number
ADRVC  r1, Buffer
MOVVC  r2, #?Buffer
SWIVC  XOS_ConvertHex2 ; Print BCD as hex
SWIVC  XOS_Write0 ; Display output
MOV    pc, lr

```

Econet_GetRegisters

Econet_GetRegisters is similar to machine peek, in that a fixed amount of information is returned from the destination machine; in this case it is 80 bytes (20 words). The registers are returned in the following order: R0 to R14, PC plus PSR, R13_irq, R14_irq, R13_svc, and R14_svc. The FIQ registers are not returned because they are used by the Econet software, and so would always be the same, and of no interest since they would reflect the state of the part of the Econet software that transmits data. It is worthwhile aligning the receive buffer for a machine peek so that each of the 20 words is on a word boundary; this makes loading them easier.

Protection against immediate operations

Because these immediate operations can be quite intrusive it is possible to prevent their reception by manipulating an internal variable of the Econet software. There is one bit in this internal variable for each operation, and you can set or clear each bit. There is also a default value for each bit which is

held in CMOS RAM. The SWI that allows you to manipulate this internal variable is Econet_SetProtection (SWI &4000E). These bits are held in a single word; the bit assignments are as follows:

Bit	Immediate operation protected against
0	Peek
1	Poke
2	Remote JSR
3	User procedure call
4	OS procedure call
5	Halt – must be zero on RISC OS computers
6	Continue – must be zero on RISC OS computers
7	Machine peek
8	Get registers
9 - 30	Reserved – must be zero.
31	Write new value to the CMOS RAM

To protect against or disable the reception of a particular immediate operation, the appropriate bit should be set in the internal variable. The SWI Econet_SetProtection call replaces the OldValue with the NewValue, The NewValue is calculated like this:

$$\text{NewValue} := (\text{OldValue AND R1}) \text{ EOR R0.}$$

Altering the protection held in CMOS RAM

When the Econet software is started up (as a result of Ctrl-Break, or *RMReInit) then the value held in CMOS RAM will be used to initialise the internal variable. To alter the value held in CMOS RAM the entry value of R0 to SWI Econet_SetProtection should have bit 31 set, which causes the resultant value to be written not only to the internal variable, but also to the CMOS RAM. Note that the use of Econet_ReadProtection (SWI &4000D) is deprecated; if you need to read the current value you should use SWI Econet_SetProtection with R0=0, and R1=&FFFFFFF.

Reading your station and network numbers

To establish what your station number is and which network you are connected to (if you have more than one), the Econet software provides a call to return these two values: Econet_ReadLocalStationAndNet (SWI &4000A). If you don't have more than one network then the network number (returned in R1) will be zero.

These values are the same as those reported by *Help Station (in fact *Help Station calls SWI Econet_ReadLocalStationAndNet to get the values).

Extracting station numbers from a string

To ensure that all Econet oriented software presents a consistent user interface there is a SWI call to read a station and/or network number from a supplied string. This call, Econet_ReadStationNumber (SWI &4000F), is used by both NetFS and NetPrint for all their command line processing. In the case of software that has a concept of a current station (and network) number the return value of -1 should mean 'use the existing value' - this is how *FS works, for example. Where there isn't a current value, as would be expected in a transient command such as *Notify, the return of -1 for the station number should be treated as an error and the return of -1 as a network number should imply the use of zero as a network number. The following is the beginning (and some of the end) of a transient command:

```
CommandStart
    LDRB    r0, [ r1 ]      ; Check the first argument exists
    TEQ    r0, #0          ; Zero means no arguments
    BEQ    SyntaxError     ; Exit with error

    SWI    XEconet_ReadStationNumber
    MOVVS  pc, lr          ; Must be able to cope
    CMP    r2, #-1        ; No station number given
    BEQ    NoStationNumberError
    CMP    r3, #-1        ; No net number given
    MOVEQ  r3, #0         ; Means use zero

    MOV    pc, lr

SyntaxError
    ADR    r0, ErrorGetRegsSyntax
    ORRS  pc, lr, #VFlag

ErrorGetRegsSyntax
    &      ErrorNumber_Syntax
    =      "Syntax: *Command <Station number>"
    =      0
    ALIGN

NoStationNumberError
    ADR    r0, ErrorUnableToDefault
    ORRS  pc, lr, #VFlag
```

```

ErrorUnableToDefault
&      ErrorNumber_UnableToDefault
=      "Either a station number or a full"
       " network address is required"
=      0
ALIGN

```

Converting station and network to a string

There exist two inverse functions that convert a station and network number pair into a string, see the section on conversions for exact details.

Conventions and values

The following conventions apply to the various values that the Econet uses:

Station numbers

Station numbers are normally in the range 1 to 254. The station number zero is used in SWI Econet_CreateReceive to indicate that reception may occur from any station. The station number 255 is used in SWI Econet_StartTransmit and in SWI Econet_DoTransmit to indicate that a broadcast is to take place; it is also used in SWI Econet_CreateReceive to indicate that reception may occur from any station, and is to be preferred over the value zero for this purpose.

Network numbers

Network numbers are normally in the range 1 to 254. The value zero means the local network; in a SWI Econet_CreateReceive it is taken to indicate that reception may occur from any network. The network number 255 is used in SWI Econet_StartTransmit and in SWI Econet_DoTransmit to indicate that a broadcast is to take place. It is also used in SWI Econet_CreateReceive to indicate that reception may occur from any station; the use of zero to indicate wild reception is deprecated.

Although RISC OS fully supports top-bit-set network numbers (ie 128 - 254), certain Econet devices – such as bridges – will not propagate them, leading to problems. You should beware of this.

Port numbers

Port numbers are normally in the range 1 to 254, although the values &90 through &9F and &D0 through &D2 are reserved by Acorn for existing protocols. Port number zero is reserved. A port number of either zero or 255 in a reception indicates that the reception may occur regardless of the port number on the incoming packet. The use of zero to indicate wild reception is deprecated.

Flag bytes

Flag byte values are in the range 0 to 255 (&FF), but only the bottom seven bits are significant.

Transmission semantics

The transmission semantics are simple. When a transmission is started the client's control information (passed in registers) is stored in a record in a linked list within Econet workspace. At regular intervals the list is scanned, and those records that should be actually transmitted at that moment are passed to the FIQ software. When that particular transmission attempt completes the status of the record is changed accordingly. This means that if two transmissions are started at the same time, they will interleave their transmission retries.

When a transmission has completed but failed:

- if the count is non-zero the delay is added to the predicted start time to give the next start time
- otherwise the status is set to *Status_NotListening* (or *Status_NetError*).

This means that as far as possible the time out time will be the Delay multiplied by the Count.

SWI Calls

Econet_CreateReceive (SWI &40000)

Creates a Receive Control Block

On entry

R0 = port number
R1 = station number
R2 = network number
R3 = buffer address
R4 = buffer size in bytes

On exit

R0 = handle
R2 = 0 if R2 on entry is the local network number

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call creates a Receive Control Block (RxCB) to control the reception of an Econet packet. It returns a handle to the RxCB.

The buffer must remain available all the time that the RxCB is open, as data received over the Econet is read directly from hardware to the buffer. You must not use memory in application space if your program is to run under the Desktop. Instead, you should use memory from the RMA.

Related SWIs

None

Related vectors

None

Econet_ExamineReceive (SWI &40001)

	Reads the status of an RxCB
On entry	R0 = handle
On exit	R0 = status
Interrupts	Interrupts are disabled Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is re-entrant
Use	This call reads the status of an RxCB, which may be one of the following: 7 Status_RxReady 8 Status_Receiving 9 Status_Received
Related SWIs	Econet_WaitForReception (SWI &40004)
Related vectors	None

Econet_ReadReceive (SWI &40002)

	Returns information about a reception, including the size of data
On entry	R0 = handle
On exit	R0 = status R1 = 0, or flag byte if R0 = 9 (Status_Received) on exit R2 = port number R3 = station number R4 = network number R5 = buffer address R6 = buffer size in bytes, or amount of data received if R0 = 9 on exit (Status_Received)
Interrupts	Interrupts are disabled Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is re-entrant
Use	This call returns information about a reception; most importantly, it tells you how much data was received, if any, and the address of the buffer in which it was placed. The buffer address is the same as that passed to Econet_CreateReceive (SWI &40000). You can call this SWI before a reception has occurred. The returned values in R3 and R4 (the network and station numbers) are those of the transmitting station if the status is Status_Received, otherwise they are the same values that were passed in to SWI Econet_CreateReceive.
Related SWIs	Econet_WaitForReception (SWI &40004)
Related vectors	None

Econet_AbandonReceive (SWI &40003)

Abandons an RxCB

On entry

R0 = handle

On exit

R0 = status

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call abandons an RxCB, returning its memory to the RMA. The reception may have completed (R0 = 9 - Status_Received - on exit), in which case the data is lost.

Related SWIs

Econet_WaitForReception (SWI &40004)

Related vectors

None

Econet_WaitForReception (SWI &40004)

Polls an RxCB, reads its status, and abandons it

On entry

R0 = handle
R1 = delay in centiseconds
R2 = 0 to ignore Escape; else Escape ends waiting

On exit

R0 = status
R1 = 0, or flag byte if R0 = 9 (Status_Received) on exit
R2 = port number
R3 = station number
R4 = network number
R5 = buffer address
R6 = buffer size in bytes, or amount of data received if R0 = 9 on exit (Status_Received)

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 repeatedly polls an RxCB (that you have already set up with Econet_CreateReceive) until a reception occurs, or a timeout occurs, or the user interferes (say by pressing Escape). It then reads the status of the RxCB before abandoning it.

The returned values in R3 and R4 (the network and station numbers) are those of the transmitting station if the status is Status_Received, otherwise they are the same values that were passed in to SWI Econet_CreateReceive.

Note that this interface enables interrupts and so can not be called from within either interrupt service code or event routines.

Related SWIs

Econet_ExamineReceive (SWI &40001), Econet_ReadReceive (SWI &40002), and Econet_AbandonReceive (SWI &40003)

Related vectors

None

Econet_EnumerateReceive (SWI &40005)

On entry	R0 = index (1 to start with first receive block)
On exit	R0 = handle (0 if no more receive blocks)
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call returns the handles of open RxCBs. On entry R0 is the number of the RxCB being asked for (1, 2, 3...). If the value of R0 is greater than the number of open RxCBs, then the value returned as the handle will be 0, which is an invalid handle.
Related SWIs	Econet_CreateReceive (SWI &40000), Econet_AbandonReceive (SWI &40003), and Econet_WaitForReception (SWI &40004)
Related vectors	None

Econet_StartTransmit (SWI &40006)

Creates a Transmit Control Block and starts a transmission

On entry

R0 = flag byte
R1 = port number
R2 = station number
R3 = network number
R4 = buffer address
R5 = buffer size in bytes (less than 8 k)
R6 = count
R7 = delay in centiseconds

On exit

R0 = handle
R1 corrupted
R2 = buffer address
R3 = station number
R4 = network number

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call creates a Transmit Control Block (TxCB) to control the transmission of an Econet packet. It then starts the transmission.

The value returned in R4 (the network number) will be the same as that passed in in R3 unless that number is equal to the local network number; in that case the network number will be returned as zero.

Related SWIs

Econet_DoTransmit (SWI &40009)

Related vectors

None

Econet_PollTransmit (SWI &40007)

Reads the status of a TxCB

On entry

R0 = handle

On exit

R0 = status

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call reads the status of a TxCB, which may be one of the following:

- 0 Status_Transmitted
- 1 Status_LineJammed
- 2 Status_NetError
- 3 Status_NotListening
- 4 Status_NoClock
- 5 Status_TxReady
- 6 Status_Transmitting

Related SWIs

Econet_DoTransmit (SWI &40009)

Related vectors

None

Econet_AbandonTransmit (SWI &40008)

	Abandons a TxCB
On entry	R0 = handle
On exit	R0 = status
Interrupts	Interrupts are disabled Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	SWI is re-entrant
Use	This call abandons a TxCB, returning its memory to the RMA.
Related SWIs	Econet_DoTransmit (SWI &40009)
Related vectors	None

Econet_DoTransmit (SWI &40009)

Creates a TxCB, polls it, reads its status, and abandons it

On entry

R0 = flag byte
R1 = port number
R2 = station number
R3 = network number
R4 = buffer address
R5 = buffer size in bytes (less than 8k)
R6 = count
R7 = delay in centiseconds

On exit

R0 = status
R1 corrupted
R2 = buffer address
R3 = station number
R4 = network number

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 creates a TxCB and repeatedly polls it until it finishes transmission, or it exceeds the count of retries. It then reads the final status of the TxCB before abandoning it.

The value returned in R4 (the network number) will be the same as that passed in in R3 unless that number is equal to the local network number; in that case the network number will be returned as zero.

Related SWIs

Econet_StartTransmit (SWI &40006), Econet_PollTransmit (SWI &40007),
and Econet_AbandonTransmit (SWI &40008)

Related vectors

None

Econet_ ReadLocalStationAndNet (SWI &4000A)

Returns a computer's station number and network number

On entry

No parameters passed in registers

On exit

R0 = station number
R1 = network number

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 returns a computer's station number and network number. The network number will be zero if there are no Econet bridges present on the network.

Related SWIs

None

Related vectors

None

Econet_ConvertStatusToString (SWI &4000B)

Converts a status to a string

On entry

R0 = status
R1 = pointer to buffer
R2 = buffer size in bytes
R3 = station number
R4 = network number

On exit

R0 = buffer
R1 = updated buffer address
R2 = updated buffer size in bytes

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 converts a status to a string held in the RISC OS ROM. This is then copied into RAM, preceded by the station and network numbers, giving a string such as:

```
Station 59.254 not listening
```

Related SWIs

Econet_ConvertStatusToError (SWI &4000C)

Related vectors

None

Econet_ConvertStatusToError (SWI &4000C)

Converts a status to a string, and then generates an error

On entry

R0 = status
R1 = pointer to error buffer
R2 = error buffer size in bytes
R3 = station number
R4 = network number

On exit

R0 = pointer to error block
V flag is set

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 converts a status to a string held in the RISC OS ROM. This is then copied into RAM, preceded by the station and network numbers, giving a string such as:

```
Station 59.254 not listening
```

Finally this call returns an error by setting the V flag, with R0 pointing to the error block.

If you use a buffer address of zero, then the string is not copied into RAM. On exit, R0 will point to the ROM string instead (which, of course, excludes the station and network numbers).

Related SWIs

Econet_ConvertStatusToString (SWI &4000B)

Related vectors

None

Econet_ReadProtection (SWI &4000D)

Reads the current protection word for immediate operations

On entry

No parameters passed in registers

On exit

R0 = current protection value

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call reads the current protection word for immediate operations. Various bits in the word, when set, disable corresponding immediate operations:

Bit	Immediate operation
0	Peek
1	Poke
2	Remote JSR
3	User procedure call
4	OS procedure call
5	Halt – must be zero on RISC OS computers
6	Continue – must be zero on RISC OS computers
7	Machine peek
8	Get registers
9 - 31	Reserved – must be zero

Note – You should preferably use the call `Econet_SetProtection` (SWI &4000E) to read the protection word instead of this call.

Related SWIs

`Econet_SetProtection` (SWI &4000E)

Related vectors

None

Econet_SetProtection (SWI &4000E)

Sets or reads the protection word for immediate operations

On entry

R0 = EOR mask word
R1 = AND mask word

On exit

R0 = old value

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 sets the protection word for immediate operations as follows:

$$\text{New value} = (\text{old value AND R1}) \text{ EOR R0}$$

Various bits in the word, when set, disable corresponding immediate operations:

Bit	Immediate operation
0	Peek
1	Poke
2	Remote JSR
3	User procedure call
4	OS procedure call
5	Halt – must be zero on RISC OS computers
6	Continue – must be zero on RISC OS computers
7	Machine peek
8	Get registers
9 - 30	Reserved – must be zero
31	Write new value to the CMOS RAM

Normally this call sets or reads the current value of the word. A default value for this word is held in CMOS RAM.

The most useful values of R0 and R1 are:

Action	R0	R1
Set current value	new value (0 - &1FF)	0
Read current value	0	&FFFFFFFF
Set default value	&80000000 + new value	0

You should use this call to read the value of the protection word, rather than Econet_ReadProtection (SWI &4000D).

Related SWIs

None

Related vectors

None

Econet_ReadStationNumber (SWI &4000F)

Extracts a station and/or network number from a supplied string

On entry

R1 = address of string to read

On exit

R1 = address of terminating space or control character

R2 = station number (-1 for not found)

R3 = network number (-1 for not found)

Interrupts

Interrupts are enabled

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call extracts a station and/or network number from a supplied string

Related SWIs

None

Related vectors

None

Econet_PrintBanner (SWI &40010)

Prints the string "Acorn Econet" followed by a newline

On entry

No parameters passed in registers

On exit

No values returned in registers

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 prints the string "Acorn Econet" followed by a newline. It calls OS_Write0 and OS_NewLine and so can not be called from within either interrupt service code or event routines.

If the Econet danetwork data clock is not present then the text " no clock" is appended to the banner.

Related SWIs

None

Related vectors

None

Econet_ReleasePort (SWI &40012)

Releases a port number that was previously claimed

On entry

R0 = port number

On exit

—

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 releases a port number that was previously claimed by calling Econet_ClaimPort (SWI &40015).

You must not use this call for port numbers that have been previously claimed using Econet_AllocatePort (SWI &40013); instead, you must call Econet_DeAllocatePort (SWI &40014).

Related SWIs

Econet_ClaimPort (SWI &40015)

Related vectors

None

Econet_AllocatePort (SWI &40013)

Allocates a unique port number

On entry

No parameters passed in registers

On exit

R0 = port number

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 allocates a unique port number that has not already been claimed or allocated.

When you have finished using the port number, you should call Econet_DeAllocatePort (SWI &40014) to make it available for use again.

Related SWIs

Econet_DeAllocatePort (SWI &40014)

Related vectors

None

Econet_DeAllocatePort (SWI &40014)

Deallocates a port number that was previously allocated

On entry

R0 = port number

On exit

—

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 deallocates a port number that was previously allocated by calling Econet_AllocatePort (SWI &40013).

You must not use this call for port numbers that have been previously claimed using Econet_ClaimPort (SWI &40015); instead, you must call Econet_ReleasePort (SWI &40012).

Related SWIs

Econet_AllocatePort (SWI &40013)

Related vectors

None

Econet_ClaimPort (SWI &40015)

On entry	R0 = port number
On exit	—
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 claims a specific port number. If it has already been claimed or allocated, an error is generated. When you have finished using the port number, you should call Econet_ReleasePort (SWI &40012) to make it available for use again.
Related SWIs	Econet_ReleasePort (SWI &40012)
Related vectors	None

Econet_StartImmediate (SWI &40016)

Creates a TxCB and starts an immediate operation

On entry

R0 = operation type
R1 = remote address or Procedure number
R2 = station number
R3 = network number
R4 = buffer address
R5 = buffer size in bytes (less than 8k)
R6 = count
R7 = delay in centiseconds

On exit

R0 = handle
R1 corrupted
R2 = buffer address
R3 = station number
R4 = network number

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call creates a TxCB and starts an immediate operation. For full details see the description in the *Technical Details* earlier in this chapter.

The value returned in R4 (the network number) will be the same as that passed in in R3 unless that number is equal to the local network number; in that case the network number will be returned as zero.

Related SWIs

Econet_DoImmediate (SWI &40017)

Related vectors

None

Econet_DoImmediate (SWI &40017)

Creates a TxCB for an immediate operation, polls it, reads its status, and abandons it

On entry

R0 = operation type
R1 = remote address or procedure number
R2 = station number
R3 = network number
R4 = buffer address
R5 = buffer size in bytes (less than 8k)
R6 = count
R7 = delay in centiseconds

On exit

R0 = status
R1 corrupted
R2 = buffer address
R3 = station number
R4 = network number

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call creates a TxCB for an immediate operation, and repeatedly polls it until it finishes transmission or it exceeds the count of retries. It then reads the final status of the TxCB before abandoning it. For full details see the description in the *Technical Details* earlier in this chapter.

The value returned in R4 (the network number) will be the same as that passed in in R3 unless that number is equal to the local network number; in that case the network number will be returned as zero.

Related SWIs

Econet_StartImmediate (SWI &40016)

Related vectors

None

* Commands

The only * Command the Econet module responds to is *Help Station, which displays the current network and station numbers of the machine. For more details of the *Help command, see the chapter entitled *The rest of the kernel*.

Hourglass

Introduction and Overview

The Hourglass module will change the pointer shape to that of an Hourglass. You can optionally also display:

- a percentage figure
- two "LED" indicators for status information (one above the Hourglass, and one below).

Note that cursor shapes 3 and 4 are used (and hence corrupted) by the Hourglass. You should not use these shapes in your programs.

Normally the Hourglass module is used to display an hourglass on the screen whenever there is prolonged activity on the Econet. The calls to do so are made by the NetStatus module, which claims the EconetV vector. See the chapters entitled *Software vectors* and *NetStatus* for further details.

The rest of this chapter details the SWIs used to control the Hourglass.

SWI Calls

Hourglass_On (SWI &406C0)

Turns on the Hourglass

On entry

—

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This turns on the Hourglass. Although control return immediately there is a delay of 1/3 of a second before the Hourglass becomes visible. Thus you can bracket an operation by Hourglass_On/Hourglass_Off so that the Hourglass will only be displayed if the operation takes longer than 1/3 of a second.

You can set a different delay using Hourglass_Start (SWI &406C3).

Hourglass_On's are nestable. If the Hourglass is already visible then a count is incremented and the Hourglass will remain visible until an equivalent number of Hourglass_Off's are done. The LEDs and percentage indicators remain unchanged.

Related SWIs

Hourglass_Off (SWI &406C1), Hourglass_Start (SWI &406C3)

Related vectors

None

Hourglass_Off (SWI &406C1)

	Turns off the Hourglass
On entry	—
On exit	—
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call decreases the count of the number of times that the Hourglass has been turned on. If this makes the count zero, it turns off the Hourglass. When the Hourglass is removed the pointer number and colours are restored to those in use at the first Hourglass_On.
Related SWIs	Hourglass_On (SWI &406C0), Hourglass_Smash (SWI &406C2)
Related vectors	None

Hourglass_Smash (SWI &406C2)

Turns off the Hourglass immediately

On entry

—

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call turns off the Hourglass immediately, taking no notice of the count of nested Hourglass_On's. If you use this call you must be sure neither you, nor anyone else, should be displaying an Hourglass.

Related SWIs

Hourglass_Off (SWI &406C1)

Related vectors

None

Hourglass_Start (SWI &406C3)

	Turns on the Hourglass after a given delay
On entry	R0 = delay before startup (in centi-seconds), or 0 to suppress the Hourglass
On exit	—
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call works in the same way as Hourglass_On, except you can specify your own startup delay. If you specify a delay of zero and the Hourglass is currently off, then future Hourglass_On and Hourglass_Start calls have no effect. The condition is terminated by the matching Hourglass_Off, or by an Hourglass_Smash.
Related SWIs	Hourglass_On (SWI &406C0), Hourglass_Off (SWI &406C1)
Related vectors	None

Hourglass_Percentage (SWI &406C4)

Displays a percentage below the Hourglass

On entry

R0 = percentage to display (if in range 0 - 99), else turns off percentage

On exit

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call controls the display of a percentage below the Hourglass. If R0 is in the range 0-99 the value is displayed; if it is outside this range, the percentage display is turned off.

The default condition of an Hourglass is not to display percentages.

Related SWIs

None

Related vectors

None

Hourglass_LEDs (SWI &406C5)

	Controls the display indicators above and below the Hourglass
On entry	R0, R1 = values used to set LEDs' word
On exit	R0 = old value of LEDs' word
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call controls the two display indicators above and below the Hourglass, which can be used to display status information. These are controlled by bits 0 and 1 respectively of the LEDs' word. The indicator is on if the bit is set, and off if the bit is clear. The new value of the word is set as follows: $\text{New value} = (\text{Old value AND R1}) \text{ XOR R0}$
	The default condition is all indicators off.
Related SWIs	None
Related vectors	None

1

NetStatus

Introduction and Overview

The NetStatus module controls the display of an hourglass on the screen whenever there is prolonged activity on the Econet.

It claims EconetV, and examines the reason for each call that is made to the vector. It in turn makes an appropriate call to the Hourglass module, so that the appearance of the Hourglass indicates the status of the net. The Hourglass has two 'LEDs', one on top and one on the bottom:

- if only the top LED is on, then your station is trying to receive
- if only the bottom LED is on, then your station is trying to transmit
- if both LEDs are on, then your station is waiting for a broadcast reply.

It also displays percentage figures (when it is able to do so meaningfully) which show the percentage of a transfer that has completed.

Technical Details

This table shows how NetStatus converts the reason codes for calls to EconetV (listed in the chapter entitled *Software vectors*) into the SWI calls that it makes to the Hourglass module:

Reason code	SWI call
NetFS_Start...	Hourglass_On
NetFS_Part...	Hourglass_Percentage
NetFS_Finish...	Hourglass_Off
NetFS_StartWait	Hourglass_LEDs (both on)
Econet_StartTransmission	Hourglass_LEDs (only top one on)
Econet_StartReception	Hourglass_LEDs (only bottom one on)
NetFS_FinishWait	Hourglass_LEDs (both off)
Econet_FinishTransmission	Hourglass_LEDs (both off)
Econet_FinishReception	Hourglass_LEDs (both off)

ColourTrans

Introduction

ColourTrans allows a program to select the physical red, green and blue colours that it wishes to use, given a particular output device and palette. ColourTrans then calculates the best colour available to fit the required colour.

Thus, an application doesn't have to be aware of the number of colours available in a given mode.

It can also intelligently handle colour usage with sprites and the font manager, and is the best way to set up colours when printing.

Before reading this chapter, you should be familiar with the VDU, sprite and font manager principles.

Overview

The ColourTrans module is currently provided in RAM in Release 2.0 of RISC OS in System:Modules.Colours, but may be moved into ROM in later releases of the OS. Any application which uses it should ensure it is present using the *RMEnsure command, say from an Obey file:

```
RMEnsure ColourTrans 0.51 RMLoad System:Modules.Colours
RMEnsure ColourTrans 0.51 Error You need ColourTrans 0.51 or later
```

Definition of terms

Here are some terms you should know when using this chapter.

GCOL is like the colour parameter passed to VDU 17. It uses a simple format for 256 colour modes.

Colour number is what is written into screen memory to achieve a given colour in a particular mode.

Palette entry is a word that contains a description of a physical colour in red, green and blue levels. Usually, this term refers to the required colour that is passed to a ColourTrans SWI.

Palette pointer is a pointer to a list of palette entries. The table would have one entry for each logical colour in the requested mode. In 256 colour mode, only 16 entries are needed, as there are only 16 palette registers.

Closest colour is the colour in the palette that most closely matches the palette entry passed. Furthest colour is the one furthest from the colour requested. These terms refer to a least-squares test of closeness.

Finding a colour

There are many SWIs that will find the best fit colour in the palette for a set of parameters. Here is a list of the different kinds of parameters that can return a best fit colour:

- Given palette entry, return nearest or furthest GCOL
- Given palette entry, return nearest or furthest colour number
- Given palette entry, mode and palette pointer, return nearest or furthest GCOL
- Given palette entry, mode and palette pointer, return nearest or furthest colour number

Setting a colour

Some SWIs will set the VDU driver GCOL to the calculated GCOL after finding it.

- Given palette entry, return nearest GCOL, and set that colour
- Given palette entry, return furthest GCOL, and set that colour

Conversion

There is a pair of SWIs to convert GCOLs to and from colour numbers. Note that this only has meaning for 256 colour modes.

Sprites and Fonts

The colour control commands in sprites and the font manager can be controlled from ColourTrans. Thus, the selection of logical colours within these modules is handled by ColourTrans, rather than an application selecting an explicit range.

Using other palette SWIs

If an application has to use other SWIs to change the palette, then there is a SWI in ColourTrans to inform it. This is because ColourTrans maintains a cache used for mapping colours. If the palette has independently changed, then it has no way of telling.

If the screen mode has changed there is no need to use this call, since the ColourTrans module detects this itself – but if output is switched to a sprite (and ColourTrans will be used) then the SWI must also be called.

Wimp

If you are using the Wimp interface, then the ColourTrans calls are fine to use, because they never modify the palette.

Printing

Because ColourTrans allows an application to request an RGB colour rather than a logical colour, it is ideal for use with the printer drivers, where a printer may be able to represent some RGB colours more accurately than the screen.

Technical Details

Colours

Two different colour systems are used in 256 colour modes. The GCOL form is much easier to use, while the colour number is optimised for the hardware. In all other colour modes, they are identical.

The palette entry used to request a given physical colour is in the same format as that used to set the anti-alias palette in the font manager.

GCOL

The 256 colour modes use a byte that looks like this:

Bit Meaning

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

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

Colour number

The 256 colour mode in the colour number looks like this:

Bit Meaning

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

In fact the bottom 4 bits of the colour number are obtained via the palette, but the default palette in 256 colour modes is set up so that the above settings apply, and this is not normally altered.

Palette entry

The palette entry is a word of the form &BBGRR00. That is, it consists of four bytes, with the palette value for the blue, green and red gun in the top three bytes. Bright white, for instance would be &FFFFFF00, while half intensity cyan would be &77770000. The current graphics hardware only uses the upper nibbles of these colours, but for upwards compatibility the lower nibble should contain a copy of the upper nibble.

Finding a colour

The SWIs that find the best fit have generally self explanatory names. As shown in the overview, they follow a standard pattern. All of the SWI names that follow are prefixed with ColourTrans_. They are as follows:

ReturnGCOL (SWI &40742)

Given palette entry, return nearest GCOL

ReturnOppGCOL (SWI &40747)

Given palette entry, return furthest GCOL

ReturnColourNumber (SWI &40744)

Given palette entry, return nearest colour number

ReturnOppColourNumber (SWI &40749)

Given palette entry, return furthest colour number

ReturnGCOLForMode (SWI &40745)

Given palette entry, mode and palette pointer, return nearest GCOL

ReturnOppGCOLForMode (SWI &4074A)

Given palette entry, mode and palette pointer, return furthest GCOL

ReturnColourNumberForMode (SWI &40746)

Given palette entry, mode and palette pointer, return nearest colour number

ReturnOppColourNumberForMode (SWI &4074B)

Given palette entry, mode and palette pointer, return furthest colour number

Palette pointers

Where a palette pointer is used, certain conventions apply:

- a palette pointer of -1 means the current palette is used
- a palette pointer of 0 means the default palette for the specified mode.

Where modes are used:

- mode -1 means the current mode.

Best fit colour

These calls use a simple algorithm to find the colour in the palette that most closely matches the high resolution colour specified in the palette entry. It calculates the *distance* between the colours, which is a weighted least squares function. If the desired colour is (R_d, B_d, G_d) and a trial colour is (R_t, B_t, G_t) , then:

$$\text{distance} = \text{redweight} * (R_t - R_d)^2 + \text{greenweight} * (G_t - G_d)^2 + \text{blueweight} * (B_t - B_d)^2$$

where redweight = 2, greenweight = 3 and blueweight = 1. These weights are set for the most visually effective solution to this problem.

Setting a colour

ColourTrans_SetGCOL (SWI &40743) will act like ColourTrans_ReturnGCOL, except that it will set the graphics system GCOL to be as close to the colour you requested as it can. Note that ECF patterns will not yet be used in monochrome modes to reflect grey shades, as they are with Wimp_SetColour.

Similarly, ColourTrans_SetOppGCOL (SWI &40748) will set the graphics system GCOL with the opposite of the palette entry passed.

Conversion

To convert between the GCOL and colour number format in 256 colour modes, the SWIs ColourTrans_GCOLToColourNumber (SWI &4074C) and ColourTrans_ColourNumberToGCOL (SWI &4074D) can be used.

Sprites and Fonts

ColourTrans_SelectTable (SWI &40740) will set up a translation table in the buffer. ColourTrans_SelectGCOLTable (SWI &40741) will set up a list of GCOLs in the buffer. See the chapter entitled *Sprites* for a definition of these tables (although the latter call does not in fact relate to sprites).

ColourTrans_ReturnFontColours (SWI &4074E) will try and find the best set of logical colours for an anti-alias colour range. ColourTrans_SetFontColours (SWI &4074F) also does this, but sets the font manager plotting colours as well. It calls Font_SetFontColours, or

Using other palette SWIs

Font_SetPalette in 256 colour modes – but it works out which logical colours to use beforehand. See the chapter entitled *The Font Manager* for details of this call and the anti-aliasing colours.

If a program has changed the palette, then ColourTrans_InvalidateCache (SWI &40750) must be called. This will reset its internal cache. This applies to Font_SetFontPalette or Wimp_SetPalette or VDU 19 or anything like that, but not to mode change, since this is detected automatically.

You must also call this SWI if output has been switched to a sprite, and ColourTrans is to be called while the output is so redirected. You must then call it again after output is directed back to the screen.

SWI Calls

ColourTrans_SelectTable (SWI &40740)

Set up a translation table in the buffer

R0 = source mode

R1 = source palette pointer

R2 = destination mode, or -1 for current mode

R3 = destination palette pointer, or -1 for current palette, or 0 for default for the mode

R4 = pointer to buffer

On entry

On exit

R0 - 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

Given a source mode and palette, and destination mode and palette, and a buffer, set up a translation table in the buffer – that is, a set of colour numbers as used by scaled sprite plotting. See the chapter entitled *Sprites* for details of such tables.

Related SWIs

None

Related vectors

ColourV

ColourTrans_SelectGCOLTable (SWI &40741)

	Set up a list of GCOLs in the buffer
On entry	R0 = source mode R1 = source palette pointer R2 = destination mode, or -1 for current mode R3 = destination palette pointer, or -1 for current palette, or 0 for default for the mode R4 = pointer to buffer
On exit	R0 - R4 preserved
Interrupts	Interrupts are enabled Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrant	SWI is not re-entrant
Use	Given a source mode and palette, and destination mode and palette, and a buffer, set up a list of GCOLs in the buffer. The values can subsequently be used by passing them to GCOL and Tint.
Related SWIs	None
Related vectors	ColourV

ColourTrans_ReturnGCOL (SWI &40742)

Get the closest GCOL for a palette entry

On entry

R0 = palette entry

On exit

R0 = GCOL

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Given a palette entry, return the closest GCOL in the current mode and palette.

This call is equivalent to ColourTrans_ReturnGCOLForMode for the given palette entry, with parameters of -1 for both the mode and palette pointer.

Related SWIs

ColourTrans_ReturnGCOLForMode (SWI &40745)

Related vectors

ColourV

ColourTrans_SetGCOL (SWI &40743)

Set the closest GCOL for a palette entry

On entry

R0 = palette entry
R3 = 0 for foreground or 128 for background
R4 = GCOL action

On exit

R0 = GCOL
R2 = \log_2 of bits-per-pixel for current mode
R3 = initial value AND &80
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

Given a palette entry, work out the closest GCOL in the current mode and palette, and set it.

The top three bytes of R3 and R4 should be zero, to allow for future expansion.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ ReturnColourNumber (SWI &40744)

Get the closest colour for a palette entry

On entry

R0 = palette entry

On exit

R0 = colour number

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Given a palette entry, return the closest colour number in the current mode and palette.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ ReturnGCOLForMode (SWI &40745)

Get the closest GCOL for a palette entry

On entry

R0 = palette entry

R1 = destination mode, or -1 for current mode

R2 = palette pointer, or -1 for current palette, or 0 for default for the mode

On exit

R0 = GCOL

R1 = preserved

R2 = preserved

Interrupts

Interrupts are enabled

Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Given a palette entry, a destination mode and palette, return the closest GCOL.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ ReturnColourNumberForMode (SWI &40746)

Get the closest colour for a palette entry

On entry

R0 = palette entry

R1 = destination mode, or -1 for current mode

R2 = palette pointer, or -1 for current palette, or 0 for default for the mode

On exit

R0 = colour number

R1 = preserved

R2 = preserved

Interrupts

Interrupts are enabled

Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Given a palette entry, a destination mode and palette, return the closest colour number.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ReturnOppGCOL (SWI &40747)

Get the furthest GCOL for a palette entry

On entry

R0 = palette entry

On exit

R0 = GCOL

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Given a palette entry, return the furthest GCOL in the current mode and palette.

This call is equivalent to ColourTrans_ReturnOppGCOLForMode for the given palette entry, with parameters of -1 for both the mode and palette pointer.

Related SWIs

None

Related vectors

ColourV

ColourTrans_SetOppGCOL (SWI &40748)

Set the furthest GCOL for a palette entry

On entry

R0 = palette entry
R3 = 0 for foreground or 128 for background
R4 = GCOL action

On exit

R0 = GCOL
R2 = \log_2 of bits-per-pixel for current mode
R3 = initial value AND &80
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

Given a palette entry, work out the furthest GCOL in the current mode and palette, and set it

The top three bytes of R3 and R4 should be zero, to allow for future expansion.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ ReturnOppColourNumber (SWI &40749)

	Get the furthest colour for a palette entry
On entry	R0 = palette entry
On exit	R0 = colour number
Interrupts	Interrupts are enabled Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	Given a palette entry, return the furthest colour number in the current mode and palette.
Related SWIs	None
Related vectors	ColourV

ColourTrans_ ReturnOppGCOLForMode (SWI &4074A)

Get the furthest GCOL for a palette entry

On entry

R0 = palette entry

R1 = destination mode or -1 for current mode

R2 = palette pointer, -1 for current palette or 0 for default for the mode

On exit

R0 = GCOL

R1 = preserved

R2 = preserved

Interrupts

Interrupts are enabled

Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Given a palette entry, a destination mode and palette, return the furthest GCOL.

Related SWIs

None

Related vectors

ColourV

ColourTrans_Return OppColourNumberForMode (SWI &4074B)

Get the furthest colour for a palette entry

On entry

R0 = palette entry
R1 = destination mode or -1 for current mode
R2 = palette pointer, -1 for current palette or 0 for default for the mode

On exit

R0 = colour number
R1 = preserved
R2 = preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Given a palette entry, a destination mode and palette, return the furthest colour number.

Related SWIs

None

Related vectors

ColourV

ColourTrans_ GCOLToColourNumber (SWI &4074C)

Translate a GCOL to a colour number

R0 = GCOL

R0 = colour number

Interrupts are enabled
Fast interrupts are enabled

Processor is in SVC mode

SWI is not re-entrant

This call will change the value passed from a GCOL to a colour number.

You should only call this SWI for 256 colour modes; the results will be meaningless for any others.

None

ColourV

On entry

On exit

Interrupts

Processor Mode

Re-entrancy

Use

Related SWIs

Related vectors

ColourTrans_ ColourNumberToGCOL (SWI &4074D)

	Translate a colour number to a GCOL
On entry	R0 = colour number
On exit	R0 = GCOL
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 change the value passed from a colour number to a GCOL. You should only call this SWI for 256 colour modes; the results will be meaningless for any others.
Related SWIs	None
Related vectors	ColourV

ColourTrans_ReturnFontColours (SWI &4074E)

Find the best range of anti-alias colours to match a pair of palette entries

On entry

R0 = font handle, or 0 for the current font
R1 = background palette entry
R2 = foreground palette entry
R3 = maximum foreground colour offset (0 - 14)

On exit

R0 = preserved
R1 = background logical colour (preserved if in 256 colour mode)
R2 = foreground logical colour
R3 = maximum sensible colour offset (up to R3 on entry)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

Given background and foreground colours and the number of anti-aliasing colours desired, this call will find the maximum range of colours that can sensibly be used. So for the given pair of palette entries, it finds the best fit in the current palette, and then inspects the other available colours to deduce the maximum possible amount of anti-aliasing up to the limit in R3.

If anti-aliasing is desirable, you should set R3 = 14 on entry; otherwise set R3 = 0 for monochrome.

The values in R1 - R3 on exit are suitable for passing to Font_SetFontColours, or including in a font string in a command (18) sequence.

Note that in 256 colour modes, you can only set 16 colours before previously returned information becomes invalid. Therefore, if you are using this SWI to obtain information to subsequently pass to the font manager, do not use more than 16 colours.

Also note that in 256 colour modes, the font manager's internal palette will be set, with all 16 entries being cycled through by ColourTrans.

See the chapter entitled *The Font Manager* for further details of the parameters used in this call.

Related SWIs

ColourTrans_SetFontColours (SWI &4074F),
Font_SetFontColours (SWI &40092)

Related vectors

ColourV

ColourTrans_SetFontColours (SWI &4074F)

Set the best range of anti-alias colours to match a pair of palette entries

On entry

R0 = font handle, or 0 for the current font
R1 = background palette entry
R2 = foreground palette entry
R3 = maximum foreground colour offset (0 - 14)

On exit

R0 = preserved
R1 = background logical colour (preserved if in 256 colour mode)
R2 = foreground logical colour
R3 = maximum sensible colour offset (up to R3 on entry)

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

For a given pair of palette entries, find the best available range of anti-alias colours in the current palette, and set the font manager to use these colours.

This SWI is equivalent to a call to ColourTrans_ReturnFontColours followed by a call to Font_SetFontColours.

Related SWIs

ColourTrans_ReturnFontColours (SWI &4074F)

Related vectors

ColourV

ColourTrans_InvalidateCache (SWI &40750)

Inform ColourTrans that the palette has been changed by some other means

On entry

—

On exit

—

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call must be issued whenever the palette has changed since ColourTrans was last called. Note that colour changes due to a mode change are detected. You only need to use this if another of the palette change operations was used.

You must also call this SWI if output has been switched to a sprite, and ColourTrans is to be called while the output is so redirected. You must then call it again after output is directed back to the screen.

As an example, the palette utility on the icon bar calls this SWI when you finish dragging one of the RGB slider bars.

Related SWIs

None

Related vectors

ColourV

1424

The Font Manager

Introduction

This chapter describes the outline font manager that is supplied with Release 1.02 of Acorn Desktop Publisher. Releases of RISC OS upto 2.00 contain an earlier version of this font manager called the **bitmap** font manager. All future releases of RISC OS will contain the outline font manager.

A *font* is a complete set of characters of a given type *style*. The font manager provides facilities for painting characters of various sizes and styles on the screen.

To allow characters to be printed in any size, descriptions of fonts can be held in files as size-independent outlines, or pre-computed at specific sizes. The font manager allows programs to request font types and sizes by name, without worrying about how they are read from the filing system or stored in memory.

The font manager also scales fonts to the desired size automatically if the exact size is not available. The fonts are, in general, proportionally spaced, and there is a facility to print justified text – that is, adjusting spaces between words to fit the text in a specified width.

An *anti-aliasing* technique can be used to print the characters. This technique uses up to 16 shades of colour to represent pixels that should only be partially filled-in. Thus, the illusion is given of greater screen resolution.

The font manager can use *hints*, which help it scale fonts to a low resolution while retaining maximum legibility.

Overview

The font manager can be divided internally into the following components:

- Find and read font files
- Cache font data in memory
- Get a handle for a font style (many commands use this handle)
- Paint a string to the VDU memory
- Change the colours that the text is painted in
- Other assorted SWIs to handle scaling and measurements

Measurement systems

Much of the font manager deals with an internal measurement system, using millipoints. This is 1/1000th of a point, or 1/72000th of an inch. This system is an abstraction from the physical characteristics of the VDU. Text can therefore be manipulated by its size, rather than in terms of numbers of pixels, which will vary from mode to mode.

OS coordinates

OS coordinates is the other system used. There are defined to be 180 OS units per inch. This is the coordinate system used by the VDU drivers, and is related to the physical pixel layout of the screen. Calls are provided to convert between these two systems, and even change the scaling factor between them.

Referencing fonts by name

A SWI is provided to scan through the list of available fonts. This allows a program to present the user with a list to select from. It is a good idea to cache this information as reading the font list with the SWI is a slow process. Another SWI will return a handle for a given font style. A handle is a byte that the font manager uses as an internal reference for the font style. This is like an Open command in a filing system. The equivalent of Close is also provided. This tells the font manager that the program has finished with the font.

There is a SWI to make a handle the currently selected one. This will be used implicitly by many calls in the font manager. It can be changed by commands within a string while painting to the VDU.

Cacheing

Cacheing is the technique of storing one or more fonts in a designated space in memory. The cacheing system decides what gets kept or discarded from its space. Two CMOS variables control how much space is used for cacheing. One sets the minimum amount, which no other part of the system will use. The other sets the maximum amount, which is the limit on what the font manager can expand the cache to.

You should adjust these settings to suit the font requirements of your application. If too little is allowed, then the system will have to continually re-load the fonts from file. If it is too large, then you will use up memory that could be used for other things.

The command *FontList is provided to show the total and used space in the cache, and what fonts are held in it. This is useful to check how the cache is occupied.

Colours

The anti-aliasing system uses up to 16 colours, depending on the screen mode. It will try, as intelligently as possible, to use these colours to shade a character giving the illusion of greater resolution.

Logical colours

The colour shades start with a background value, which is usually the colour that the character is painted onto. They progress up to a foreground colour, which is the desired colour for the character to appear in. This is usually what appears in the centre of the character. Both of these can be set to any valid logical colour numbers.

Palette

In between background and foreground colours can be a number of other logical colours. There is a call to program the palette so that these are set to graduating intermediate levels. The points of transition are called thresholds. The thresholds are set up so that the gradations produce a smooth colour change from background to foreground.

Painting

A string can be painted into the VDU memory. As well as printable characters which are displayed in the current font style, there are non-printing command characters, used in much the same way as those in the VDU driver. They can perform many operations, such as:

- changing the colour
- altering the write position in the x and y axes
- changing the font handle
- changing the appearance and position of the underlining

By using these command characters, a single string can be displayed with as many changes of these characteristics as required

Measuring

Many SWIs exist to measure various attributes of fonts and strings. With a font, you can determine the largest box needed to contain any character in the set. This is called its bounding box. You can also check the bounding box of an individual character.

With a string, you can measure its bounding box, or check where in the string the caret would be for a given coordinate. The caret is a special cursor used with fonts. It is usually displayed as a vertical bar with twiddles on each end.

VDU calls

A number of font manager operations can be performed through VDU commands. These have been kept for compatibility and you should not use them, as they may be phased out in future versions.

Technical Details

An easy way to introduce you to programming with the font manager is to use a simple example. It shows how to paint a text string on the screen using font manager SWIs. Further on in this section is a more detailed explanation of these and all other font SWIs.

Here is the sequence that you would use:

- Font_FindFont – to 'open' the font in the size required
- Font_SetFont – to make it the currently selected font and size
- Font_SetPalette – to set the range of colours to use
- Font_Paint – to paint the string on the screen
- Font_LoseFont – to 'close' the font

Measurement systems

Internal coordinates

The description of character and font sizes comes from specialist files called metrics files. The numbers in these files are held in units of 1/1000th of an em. An em is the size of a point multiplied by the the point size of the font. For example, in a 10 point font, an em is 10 points, while in a 14 point font it is 14 points. The font manager converts 1000ths of ems into 1000ths of points, or millipoints, to use for its internal coordinate system. A millipoint is equal to 1/72000th of an inch. This has the advantage that rounding errors are minimal, since coordinates are only converted for the screen at the last moment. It also adds a level of abstraction from the physical characteristics of the target screen mode.

OS coordinates

Unfortunately, the coordinates provided for plot calls are only 16 bits, so this would mean that text could only be printed in an area of about 6/7ths of an inch.

Therefore, the font painter takes its initial coordinates from the user in the same coordinates as the screen uses, which are known as OS units. To make the conversion from OS units to points, the font painter assumes by default that there are 180 OS units to the inch. You can read and set this scale factor, which you may find useful to accurately calibrate the on screen fonts, or to build high resolution bitmaps.

Internal resolution

When the font painter moves the graphics point after printing a character, it does this internally to a resolution of millipoints, to minimise the effect of cumulative errors. The font painter also provides a justification facility, to save you the trouble of working the positions out yourself. The application can obtain the widths of characters to a resolution of millipoints.

SWIs

A pair of routines can be used to convert to and from internal millipoint coordinates to the external OS coordinates. `Font_ConverttoOS` (SWI &40088) will go from millipoints, while `Font_Converttopoints` (SWI &40089) will go to them.

Scaling factor

The scaling factor that the above SWIs (and many others in the font manager) use can be read with `Font_ReadScaleFactor` (SWI &4008F), or set with `Font_SetScaleFactor` (SWI &40090).

Font files

The font files relating to a font are all contained in a single directory:

Filename	Contents
<code>IntMetrics</code>	metrics information (character widths etc)
<code>x90y45</code>	old format pixel file (4-bits-per-pixel)
<code>f9999x9999</code>	new format pixel file (4-bits-per-pixel)
<code>b9999x9999</code>	new format pixel file (1-bits-per-pixel)
<code>Outlines</code>	new format outline file

The '9999's referred to above mean 'any decimal number in the range 1-9999'. They refer to the pixel size of the font contained within the file, which is equal to:

$$(\text{font size in } 1/16\text{ths of a point}) * \text{dots per inch} / 72$$

so, for example, a file containing 4-bits-per-pixel 12 point text at 90 dots per inch would be called `f240x240`, because $12 * 16 * 90 / 72 = 240$.

The formats of these files are detailed in the appendix entitled *File formats*.

The minimal requirement for a font is that it should contain an `IntMetrics` file and an `x90y45` or `Outlines` file. In addition, it can have any number of `f9999x9999` or `b9999x9999` files, to speed up the cacheing of common sizes.

Master and slave fonts

If outline data or scaled 4-bpp data is to be used as the source of font data it is first loaded into a 'master' font in the cache, which can be shared between many 'slave' fonts at various sizes. There can be only one master font for a given font name, regardless of size, whereas each size of font requires a separate slave font. If the data is loaded directly from the disc into the slave font, the master font is not required.

Referencing fonts by name

The font manager uses the path variable Font\$Path when it searches for fonts. This contains a list of filename prefixes which are, in turn, placed before the requested font name. The font manager uses the first directory that matches, provided it also contains an IntMetrics file. Because the variable is a list of path names, you can keep separate libraries of fonts.

The old font manager used the variable Font\$Prefix to specify a single font directory. For compatibility, the font manager looks when it is initialised to see if Font\$Path has been defined – if not, it initialises it as follows:

```
*SetMacro Font$Path <Font$Prefix>.
```

This ensures that the old Font\$Prefix directory is searched if you haven't explicitly set up the font manager to look elsewhere. The '.' on the end is needed, as Font\$Path is a prefix rather than a directory name.

*FontCat will list all the fonts that can be found using the path variable.

Changing the font path

Applications which allow the user access to fonts should call Font_ListFonts repeatedly to discover the list of fonts available. This is normally done when the program starts up.

However, it often happens that families of fonts are to be found in separate font "application" directories, whose !Run file RMEnsures the correct font manager module from within itself and then either adds itself to Font\$Path or resets Font\$Path and Font\$Prefix so that it is the only directory referenced.

In order to ensure that the user can access the new fonts available, applications should check whether the value of Font\$Path or Font\$Prefix has changed since the list of fonts was last cached, and recache the list if so. A BASIC program could accomplish this as follows:

```

size% = 4200
DIM buffer% size%      : REM this could be a scratch buffer

...

SYS "OS_GSTrans", "<Font$Prefix> and <Font$Path>", buffer%, size%-1 TO ,, length%
buffer%?length% = 13   :REM ensure there is a terminator (13 for BASIC)
IF $buffer%<>oldfontpath$ THEN
    oldfontpath$ = $buffer%
    PROCcache_list_of_fonts
ENDIF

```

Note that if the buffer overflows the string is simply truncated, so it is possible that the check may miss some changes to Font\$Prefix. However, since new elements are normally added to the front of Font\$Path, this will probably not matter.

The application could scan the list of fonts when it started up, remembering the value of Font\$Path and Font\$Prefix in oldfontpath\$, and then make the check described above just before the menu tree containing the list of fonts was about to be opened.

Alternatively the application could scan the list of fonts only when required, by setting oldfontpath\$="" when it started up, and checking for Font\$Path changing only when the font submenu is about to be opened (using the Message_MenuWarning message protocol.)

Opening and closing a font

In order to use a font, Font_FindFont (SWI &40081) must be used. This returns a handle for the font, and can be considered conceptually like a file open. In order to close it, Font_LoseFont (SWI &40082) must be used.

Handles

Font_ReadDefn (SWI &40083) will read the description of a handle, as it was created with Font_FindFont.

In order for a handle to be used, it should be set as the current handle with Font_SetFont (SWI &4008A). This setting stays until changed by another call to this function, or while painting, by a character command to change the handle.

Font_CurrentFont (SWI &4008B) will tell you what the handle of the currently selected font is.

Cacheing

Setting cache size

The size of the cache can be set with two commands. *Configure FontSize sets the minimum that will be reserved. This allocation is protected by RISC OS and will not be used for any other purpose. Running the Task Display from the desktop and sliding the bar for font cache will change this setting until the next reset.

Above this amount, *Configure FontMax sets a maximum amount of memory for font cacheing. The difference between FontSize and FontMax is taken from unallocated free memory as required to accomodate fonts currently in use. If other parts of the system have used up all this memory, then fonts will be limited to FontSize. If there is plenty of free unallocated memory, then FontMax will stop font requirements from filling up the system with cached fonts.

Cache size

*FontList will generate a list of the size and free space of the cache, as well as a list of the fonts currently cached. Font_CacheAddr (SWI &40080) can be used in a program to get the cache size and free space.

Font_LoseFont

When a program calls Font_LoseFont, the font may not be discarded from memory. The cacheing system decides when to do this. A usage count is kept, so that it knows when no task is currently using it. An 'age' is also kept, so that the font manager knows when it hasn't been used for some time.

Colours

Colour selection with the font manager involves the range of logical colours that are used by the anti-aliasing software and the physical colours that are displayed.

Logical colours

The logical colour range required is set by Font_SetFontColours (SWI &40092). This sets the background colour, the foreground colour and the range of colours in between.

Physical colours

Font_SetPalette (SWI &40093) duplicates what Font_SetFontColours does, and uses two extra parameters. These specify the foreground and background physical colours, using 4096 colour resolution. Given a range of logical colours and the physical colours for the start and finish of them, this SWI will program the palette with all the intermediate values.

Wimp environment

It must be strongly emphasised that if the program you are writing is going to run under the wimp environment then you must not use Font_SetPalette. It will damage the wimp's colour information. It is better to use Wimp_SetFontColours (SWI &400F3) or ColourTrans_SetFontColours (SWI &4074F) to use colours that are already in the palette.

Thresholds

The setting of intermediate levels uses threshold tables. These can be read with Font_ReadThresholds (SWI &40094) or set with Font_SetThresholds (SWI &40095). They use a lookup table that is described in Font_ReadThresholds.

Painting

Font_Paint (SWI &40086) is the central SWI that puts text onto the screen. It commences painting with the current handle, set with Font_SetFont. Printable characters it displays appropriately, using the current handle. The embedded character commands are as follows:

Number	Effect
9	x coordinate change in millipoints
11	y coordinate change in millipoints
17	change foreground or background colour
18	change foreground, background and range of colours
21	comment string that is not displayed
25	change underline position and thickness
26	change font handle

Note that these are not compatible with VDU commands. Any non-printing characters not in the above list will generate an error, apart from 0, 10 and 13 (which are the only valid terminators).

Measuring

When reading about these measuring calls, take particular notice of the units of measurement, because they are not the same for all SWIs.

Font and character size

Font_ReadInfo (SWI &40084) will find the bounding box in pixels for a font – the maximum area used by any character within the font. Font_CharBBox (SWI &4008E) will get the bounding box for a particular character in a font. This can be in OS coordinates or millipoints.

String size

Font_StringBBox (SWI &40097) will measure the bounding box of a string in millipoints without actually printing it.

Font_StringWidth (SWI &40085) performs a similar function, but with more control. You pass it a maximum x and y size in millipoints and a character to split the string on, and it works out where to break it. It returns the size in millipoints and the index into the string at the break point.

After using this SWI, you can call Font_FutureFont (SWI &4008C). This will return what the font and colours would be if the string was passed through Font_Paint.

Caret

If the pointer is clicked on a string, and the caret needs to be placed on a character, it is necessary to calculate where on the string it would be. Font_FindCaret (SWI &4008D) will do this, though it must be passed coordinates in millipoints offset from the base of the string. Font_FindCaretJ (SWI &40096) performs the same function as Font_FindCaret, except that it compensates for a justified string.

You can plot the caret at a given height, position and colour using Font_Caret (SWI &40087). Its height should be adjusted to suit the point size of the font it is placed with. The information returned from Font_ReadInfo would be appropriate for this adjustment.

Mixing fonts' metrics and characters

Where you are using an external printer (eg. PostScript) which has a larger range of fonts than those available on the screen, it can often be useful to use a similar-looking font on the screen, using the appropriate metrics (ie. spacing) for the printer font.

The font manager provides a facility whereby a font can be created which has its own IntMetrics file, matching the appropriate font on the printer, but uses another font's characters on the screen.

This is done by putting a file called 'Outlines' in the font's directory which simply contains the name of the appropriate screen font to use. The font manager will use the IntMetrics file from the font's own directory, but will look in the other font's directory for any bitmap or outline information.

SWI Calls

Font_CacheAddr (SWI &40080)

Get the version number, font cache size and amount used

On entry

—

On exit

R0 = version number
R2 = total size of font cache (bytes)
R3 = amount of font cache used (bytes)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The version number returned is the actual version multiplied by 100. For example, version 2.42 would return 242.

This call also returns the font cache size and the amount of space used in it.

*FontList can be used to display the font cache size and space.

Related SWIs

None

Related vectors

None

Font_FindFont (SWI &40081)

Get the handle for a font

On entry

R1 = pointer to font name (terminated by a Ctrl char)
R2 = x point size * 16 (ie. in 1/16ths point)
R3 = y point size * 16 (ie. in 1/16ths point)
R4 = x resolution in dots per inch (0 = use default)
R5 = y resolution in dots per inch (0 = use default)

On exit

R0 = font handle
R1 = preserved
R2 = preserved
R3 = preserved
R4 = x resolution in dots per inch
R5 = y resolution in dots per inch

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 returns a handle to a font whose name, point size and screen resolution are passed. It also sets it as the current font, to be used for future calls to Font_paint etc.

The VDU command:

```
VDU 23,26,<font handle>,<pt size>,<x dpi>,<y dpi>,<x scale>,<y scale>,0,0,<font name>
```

is an equivalent command to this SWI. As with all VDU font commands, it has been kept for compatibility with earlier versions of the operation system and must not be used.

Related SWIs

Font_LoseFont (SWI &40082)

Related vectors

None

Font_LoseFont (SWI &40082)

	Finish use of a font
On entry	R0 = font handle
On exit	R0 = 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 tells the font manager that a particular font is no longer required.
Related SWIs	Font_FindFont (SWI &40081)
Related vectors	None

Font_ReadDefn (SWI &40083)

Read details about a font

On entry

R0 = font handle
R1 = pointer to buffer to hold font name

On exit

R1 = pointer to buffer (now contains font name)
R2 = x point size * 16 (ie in 1/16ths point)
R3 = y point size * 16 (ie in 1/16ths point)
R4 = x resolution (dots per inch)
R5 = y resolution (dots per inch)
R6 = age of font
R7 = usage count of font

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 returns a number of details about a font. The usage count gives the number of times that `Font_FindFont` has found the font, minus the number of times that `Font_LoseFont` has been used on it. The age is the number of font accesses made since this one was last accessed.

Note that the x resolution in a 132 column mode will be the same as an 80 column mode. This is because it is assumed that it will be used on a monitor that displays it correctly, which is not the case with all monitors.

Related SWIs

None

Related vectors

None

Font_ReadInfo (SWI &40084)

	Get the font bounding box
On entry	R0 = font handle
On exit	R1 = minimum x coordinate in OS units for the current mode (inclusive) R2 = minimum y coordinate in OS units for the current mode (inclusive) R3 = maximum x coordinate in OS units for the current mode (exclusive) R4 = maximum y coordinate in OS units for the current mode (exclusive)
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 returns the minimal area covering any character in the font. This is called the font bounding box.
Related SWIs	Font_CharBBox (SWI &4008E), Font_StringBBox (SWI &40097)
Related vectors	None

Font_StringWidth (SWI &40085)

Calculate how wide a string would be

On entry

R1 = pointer to string
R2 = maximum x offset before termination in millipoints
R3 = maximum y offset before termination in millipoints
R4 = 'split' character (-1 for none)
R5 = index of character to terminate by

On exit

R1 = pointer to character where the scan terminated
R2 = x offset after printing string (up to termination)
R3 = y offset after printing string (up to termination)
R4 = no of 'split' characters in string (up to termination)
R5 = index into string giving point at which the scan terminated

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call is used to calculate how wide a string would be.

The 'split' character is one at which the string can be split if any of the limits are exceeded. If R4 contains -1 on entry, then on exit it contains the number of printable (as opposed to 'split') characters found.

The string is allowed to contain command sequences, including font-change (26,) and colour-change (17,<colour>). After the call, the current font foreground and background call are unaffected, but a call can be made to Font_FutureFont to find out what the current font would be after a call to Font_Paint.

The string width function terminates as soon as R2, R3 or R5 are exceeded, or the end of the string is reached. It then returns the state it had reached, either:

- just before the last 'split' char reached
- if the 'split' char is -1, then before the last char reached
- if R2, R3 or R5 are not exceeded, then at the end of the string.

By varying the entry parameters, the string width function can be used for any of the following purposes:

- finding the cursor position in a string if you know the coordinates (although `Font_FindCaret` is better for this)
- finding the cursor coordinates if you know the position
- working out where to split lines when formatting (set R4=32)
- finding the length of a string (eg. for right-justify)
- working out the data for justification (as the font manager does).

Related SWIs

`Font_FutureFont` (SWI &4008C)

Related vectors

None

Font_Paint (SWI &40086)

Write a string to the screen

On entry

R1 = pointer to string
R2 = plot type
R3 = x coordinate (in OS coordinates or millipoints)
R4 = y coordinate (in OS coordinates or millipoints)

On exit

R1 = preserved
R2 = preserved
R3 = preserved
R4 = 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

The plot type is given by the bits of R2 as follows:

Bit	Action if set
0	1 = justify text, 0 = left justify
1	1 = rub-out box required, 0 = no box
2	reserved – must be zero
3	reserved – must be zero
4	1 = OS coordinates supplied, 0 = millipoints
5 - 7	reserved – must be zero

To justify text, you must supply a right hand side position, by previously calling a VDU 25 move command. To use the rub-out box, you must have called VDU 25 move twice, to describe the rectangle to clear: first the lower-left coordinate (which is inclusive), then the upper-right coordinate (which is exclusive). Thus, to use both, three VDU 25 moves must be made, with the justify position being last.

The string is allowed to contain characters that act as commands, like the VDU sequences:

- 9,<dx low>,<dx middle>,<dx high>
- 11,<dy low>,<dy middle>,<dy high>
- 17,<foreground colour> (+&80 for background colour)
- 18,<background>,<foreground>,
- 21,<comment string>,<terminator (any Ctrl char)>
- 25,<underline position>,<underline thickness>
- 26,

After the call, the current font and colours are updated to the last values set by command characters.

Characters 9 and 11 allow for movement within a string. This is useful for printing superscripts and subscripts, as well as tabs, in some cases. They are each followed by a 3-byte sequence specifying a number (low byte first, last byte sign-extended), which is the amount to move by in millipoints. Subsequent characters are plotted from the new position onwards.

An example of moving in the Y direction (character 11) would look like the following example, where `chr()` is a function that converts a number into a character and `move` is the movement in millipoints:

```
MoveString = chr(11)+chr(move AND &FF)+
             chr((move AND &FF00) >> 8)+
             chr((move AND &FF0000) >> 16)
```

Character 17 will act as if the foreground or background parameters passed to `Font_SetFontColours` (SWI &40092) had been changed. Character 18 allows all three parameters to that SWI to be set. See that SWI for a description of these parameters.

The *underline position* following a character 25 is the position of the top of the underline relative to the baseline of the current font, in units of 1/256th of the current font size. It is a sign-extended 8 bit number, so an underline below the baseline can be achieved by setting the underline position to a value greater than 127. The *underline thickness* is in the same units, although it is not sign-extended.

Note that when the underline position and height are set up, the position of the underline remains unchanged thereafter, even if the font in use changes. For example, you do not want the thickness of the underline to change just because some of the text is in italics. If you actually want the thickness of the underline to change, then another underline-defining sequence must be inserted at the relevant point. Note that the underline is always printed in the same colour as the text, and that to turn it off you must set the underline thickness to zero.

The VDU command VDU 25,&D0-&D7,<x coordinate>,<y coordinate>,<text string> is an equivalent command to this SWI. As with all VDU font commands, it has been kept for compatibility with earlier versions of the operation system and must not be used.

Related SWIs

Font_StringWidth (SWI &40085)

Related vectors

None

Font_Caret (SWI &40087)

	Define text cursor for font manager
On entry	R0 = colour (exclusive ORed onto screen) R1 = height (in OS coordinates) R2 bit 4 = 0 \Rightarrow R3, R4 in millipoints = 1 \Rightarrow R3, R4 in OS coordinates R3 = x coordinate (in OS coordinates or millipoints) R4 = y coordinate (in OS coordinates or millipoints)
On exit	R0 = preserved R1 = preserved R2 = preserved R3 = preserved R4 = 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	The 'caret' is a symbol used as a text cursor when dealing with anti-aliased fonts. The height of the symbol, which is a vertical bar with 'twiddles' on the end, can be varied to suit the height of the text, or the line spacing. The colour is in fact Exclusive ORed onto the screen, so in 256-colour modes it is equal to the values used in a 256-colour sprite.
Related SWIs	None
Related vectors	None

Font_ConverttoOS (SWI &40088)

Convert internal coordinates to OS coordinates

On entry

R1 = x coordinate (in millipoints)
R2 = y coordinate (in millipoints)

On exit

R1 = x coordinate (in OS units)
R2 = y coordinate (in OS units)

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 converts a pair of coordinates from millipoints to OS units, using the current scale factor. (The default is 400 millipoints per OS unit.)

Related SWIs

Font_Converttopoints (SWI &40089), Font_ReadScaleFactor (SWI &4008F),
Font_SetScaleFactor (SWI &40090)

Related vectors

None

Font_Converttopoints (SWI &40089)

Convert OS coordinates to internal coordinates

On entry

R1 = x coordinate (in OS units)
R2 = y coordinate (in OS units)

On exit

R0 is corrupted
R1 = x coordinate (in millipoints)
R2 = y coordinate (in millipoints)

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 converts a pair of coordinates from OS units to millipoints, using the current scale factor. (The default is 400 millipoints per OS unit.)

Related SWIs

Font_ConverttoOS (SWI &40088), Font_ReadScaleFactor (SWI &4008F),
Font_SetScaleFactor (SWI &40090)

Related vectors

None

Font_SetFont (SWI &4008A)

	Select the font to be subsequently used
On entry	R0 = handle of font to be selected
On exit	R0 = 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 sets up the font which is used for subsequent painting or size-requesting calls (unless overridden by a command 26, sequence in a string passed to Font_Paint).
Related SWIs	Font_SetFontColours (SWI &40092), Font_CurrentFont (SWI &4008B)
Related vectors	None

Font_CurrentFont (SWI &4008B)

	Get current font handle and colours
On entry	–
On exit	R0 = handle of currently selected font R1 = current background logical colour R2 = current foreground logical colour R3 = foreground colour offset
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	<p>This call returns the state of the font manager's internal characteristics which will apply at the next call to <code>Font_Paint</code>.</p> <p>The value in R3 gives the number of colours that will be used in anti-aliasing. The colours are $f, f+1 \dots f+\text{offset}$, where f is the foreground colour returned in R2, and offset is the value returned in R3. This can be negative, in which case the colours are $f, f-1 \dots f- \text{offset}$. Negative offsets are useful for inverse anti-aliased fonts.</p> <p>Offsets can range between -14 and $+14$. This gives a maximum of 15 foreground colours, plus one for the font background colour. If the offset is 0, just two colours are used: those returned in R1 and R2.</p> <p>The font colours, and number of anti-alias levels, can be altered using <code>Font_SetFontColours</code>, <code>Font_SetPalette</code>, <code>Font_SetThresholds</code> and <code>Font_Paint</code>.</p>
Related SWIs	<code>Font_SetFont</code> (SWI &4008A), <code>Font_SetFontColours</code> (SWI &40092), <code>Font_SetPalette</code> (SWI &40093), <code>Font_SetThresholds</code> (SWI &40095), <code>Font_Paint</code> (SWI &40086)
Related vectors	None

Font_FutureFont (SWI &4008C)

Check font characteristics after Font_StringWidth

On entry

-

On exit

R0 = handle of font which would be selected

R1 = future background logical colour

R2 = future foreground logical colour

R3 = foreground colour offset

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 can be made after a Font_StringWidth to discover the font characteristics after a call to Font_Paint, without actually having to paint the characters.

Related SWIs

Font_StringWidth (SWI &40085), Font_Paint (SWI &40086)

Related vectors

None

Font_FindCaret (SWI &4008D)

	Find where the caret is in the string
On entry	R1 = pointer to string R2 = x offset in millipoints R3 = y offset in millipoints
On exit	R1 = pointer to character where the search terminated R2 = x offset after printing string (up to termination) R3 = y offset after printing string (up to termination) R4 = number of printable characters in string (up to termination) R5 = index into string giving point at which it terminated
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	On exit, the registers give the nearest point in the string to the caret position specified on entry. This call effectively makes two calls to Font_StringWidth to discover which character is nearest the caret position. It is recommended that you use this call, rather than perform the calculations yourself using Font_StringWidth, though this is also possible.
Related SWIs	Font_StringWidth (SWI &40085), Font_FindCaret] (SWI &40096)
Related vectors	None

Font_CharBBox (SWI &4008E)

Get the bounding box of a character

On entry

R0 = font handle
R1 = ASCII character code
R2 = flags (bit 4 set ⇒ return OS coordinates, else millipoints)

On exit

R1 = minimum x of bounding box (inclusive)
R2 = minimum y of bounding box (inclusive)
R3 = maximum x of bounding box (exclusive)
R4 = maximum y of bounding box (exclusive)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

You can use this call to discover the bounding box of any character from a given font. If OS coordinates are used and the font has been scaled, the box may be surrounded by an area of blank pixels, so the size returned will not be exactly accurate. For this reason, you should use millipoints for computing, for example, line spacing on paper. However, the millipoint bounding box is not guaranteed to cover the character when it is painted on the screen, so the OS unit bounding box should be used for this purpose.

Related SWIs

Font_ReadInfo (SWI &40084), Font_StringBBox (SWI &40097)

Related vectors

None

Font_ReadScaleFactor (SWI &4008F)

	Read the internal to OS conversion factor
On entry	-
On exit	R1 = x scale factor R2 = y scale factor
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	The x and y scale factors are the numbers used by the font manager for converting between OS coordinates and millipoints. The default value is 400 millipoints per OS unit. This call allows the current values to be read.
Related SWIs	Font_ConverttoOS (SWI &40088), Font_SetScaleFactor (SWI &40090), Font_Converttopoints (SWI &40089)
Related vectors	None

Font_SetScaleFactor (SWI &40090)

Set the internal to OS conversion factor

On entry

R1 = x scale factor

R2 = y scale factor

On exit

R1 = preserved

R2 = 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

Applications that run under the Desktop should not use this call, as other applications may be relying on the current settings. If you must change the values, you should read the current values beforehand, and restore them afterwards. The default value is 400 millipoints per OS unit.

Related SWIs

Font_ConverttoOS (SWI &40088), Font_ReadScaleFactor (SWI &4008F),

Font_Converttopoints (SWI &40089)

Related vectors

None

Font_ListFonts (SWI &40091)

Scan for fonts, returning their names one at a time

On entry

R1 = pointer to 40-byte buffer for font name
R2 = count (0 on first call)
R3 = pointer to path string, or -1 to use Font\$Path

On exit

R1 = preserved
R2 = updated (-1 if no more names)
R3 = 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 searches the path pointed to (or given by the variable Font\$Path if R3 = -1), and its sub-directories, for files ending in '.IntMetrics'. When such a file is found, the full name of the subdirectory is put in the buffer, terminated by a carriage return. If the same font name is found via different path elements, only the first one will be reported.

It is started by passing a zero in R2, and indicates the end of the list by returning a -1 in R2.

The font manager command *FontCat calls this SWI internally.

Related SWIs

None

Related vectors

None

Font_SetFontColours (SWI &40092)

Change the current colours and (optionally) the current font

On entry

R0 = font handle (0 for current font)
R1 = background logical colour
R2 = foreground logical colour
R3 = foreground colour offset (-14 to +14)

On exit

R0 = preserved
R1 = preserved
R2 = preserved
R3 = 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 is used to set the current font (or leave it as it is), and change the logical colours used. In up to 16 colour modes, the three registers are used as follows:

- R1 is the logical colour of the background
- R2 is the logical colour of the first foreground colour to use
- R3 specifies the offset from the first foreground colour to the last, which is used as the actual foreground colour.

The range specified must not exceed the number of logical colours available in the current screen mode, as follows:

Colours in mode	Possible values of R1,R2,R3 to use all colours
2	0,1,0
4	0,1,2
16 or 256	0,1,14

In a 16 colour mode, to use the top 8 colours, which are normally flashing colours, the values 8,9,6 could be used.

Note that 16 is the maximum number of anti-alias colours. In 256-colour modes, the background colour is ignored, and the foreground colour is taken as an index into a table of pseudo-palette entries – see `Font_SetPalette`.

Related SWIs

`Font_SetFont` (SWI &4008A), `Font_CurrentFont` (SWI &4008B),
`Font_SetPalette` (SWI &40093)

Related vectors

None

Font_SetPalette (SWI &40093)

Define the anti-alias palette

On entry

R1 = background logical colour
R2 = foreground logical colour
R3 = foreground colour offset
R4 = physical colour of background
R5 = physical colour of last foreground

On exit

R1 = preserved
R2 = preserved
R3 = preserved
R4 = preserved
R5 = 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 sets the anti-alias palette. The values in R1, R2 and R3 have the same use as in `Font_SetFontColours`. See the description of that SWI on the previous page for the use of these parameters.

Before describing the use of R4 and R5, it must be strongly emphasised that if the program you are writing is going to run under the wimp environment then you must not use this call. It will damage the wimp's colour information. You must instead choose from the range of colours already available by using `Wimp_SetFontColours` (SWI &400E6) or `ColourTrans_SetFontColours` (SWI &400E4) instead.

R4 and R5 contain physical colour setting information. R4 describes the background colour and R5 the foreground colour. The foreground colour is the dominant colour of the text and generally appears in the middle of each

character. This SWI will set the palette colour for the range described in R1, R2 and R3 using R4 and R5 to describe the colours at each end. It will set the intermediate colours incrementally between those of R4 and R5.

The physical colours in R4 and R5 are of the form &BBGGRR00. That is, it consists of four bytes, with the palette entry for the blue, green and red gun in the upper bytes. Bright white, for instance, would be &FFFFFF00, while half intensity cyan is &77770000. The current graphics hardware only uses the upper nibbles of these colours, but for upwards compatibility the lower nibble should contain a copy of the upper nibble.

In non-256-colour modes, the palette is programmed so that there is a linear progression from the colour given in R4 to that in R5

The VDU command: VDU 23,25,&80+<background logical colour>, <foreground logical colour>,<start R>,<start G>,<start B>,<end R>,<end G>,<end B> is an equivalent command to this SWI. As with all VDU font commands, it has been kept for compatibility with earlier versions of the operation system and must not be used.

Related SWIs

Font_SetFontColours (SWI &40092)

Related vectors

None

Font_ReadThresholds (SWI &40094)

Read the list of threshold values for painting

On entry

R1 = pointer to result buffer

On exit

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 reads the list of threshold values that the font manager uses when painting characters. Fonts are defined using up to 16 anti-aliased levels. The threshold table gives a mapping from these levels to the logical colours actually used to paint the character.

The format of the data read is:

Offset	Value
0	Foreground colour offset
1	1st threshold value
2	2nd threshold value
3	:
n	&FF

The table is used in the following way. Suppose you want to use eight colours for anti-aliased colours, one background colour and seven foreground colours. Thus the foreground colour offset is 6 (there are 7 colours). The table would be set up as follows:

Offset	Value
0	6
1	2
2	4
3	6
4	8
5	10
6	12
7	14
8	&FF

When this has been set-up (using `Font_SetThresholds`), the mapping from the 16 colours to the eight available will look like this:

Input	Output	Threshold
0	0	
1	0	
2	1	2
3	1	
4	2	4
5	2	
6	3	6
7	3	
8	4	8
9	4	
10	5	10
11	5	
12	6	12
13	6	
14	7	14
15	7	

Where the output colour is 0, the font background colour is used. Where it is in the range 1-7, the colour $f+o-1$ is used, where 'f' is the font foreground colour, and 'o' is the output colour.

You can view the thresholds as the points at which the output colour 'steps up' to the next value.

Related SWIs

Font_SetThresholds (SWI &40095), Font_SetPalette (SWI &40093),
Font_SetFontColours (SWI &40092)

Related vectors

None

Font_SetThresholds (SWI &40095)

Defines the list of threshold values for painting

On entry

R1 = pointer to threshold data

On exit

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 sets up the threshold table for a given number of foreground colours. The format of the input data, and its interpretation, is explained in the previous section.

This command should rarely be needed, because the default set will work well in most cases.

The VDU command VDU 23,25,<bits per pixel>,<threshold 1>,...,<threshold 7> is an equivalent command to this SWI. As with all VDU font commands, it has been kept for compatibility with earlier versions of the operation system and must not be used.

Related SWIs

Font_ReadThresholds (SWI &40094), Font_SetPalette (SWI &40093),
Font_SetFontColours (SWI &40092)

Related vectors

None

Font_FindCaretJ (SWI &40096)

Find where the caret is in a justified string

On entry

R1 = pointer to string
R2 = x offset in millipoints
R3 = y offset in millipoints
R4 = x justification offset
R5 = y justification offset

On exit

R1 = pointer to character where the search terminated
R2 = x offset after printing string (up to termination)
R3 = y offset after printing string (up to termination)
R4 = no of printable characters in string (up to termination)
R5 = index into string giving point at which it terminated

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The 'justification offsets', R4 and R5, are calculated by dividing the extra gap to be filled by the justification of the number of spaces (ie character 32) in the string. If R4 and R5 are both zero, then this call is exactly the same as Font_FindCaret.

Related SWIs

Font_FindCaret (SWI &4008D)

Related vectors

None

Font_StringBBox (SWI &40097)

	Measure the size of a string
On entry	R1 = pointer to string
On exit	R1 = bounding box minimum x in millipoints (inclusive) R2 = bounding box minimum y in millipoints (inclusive) R3 = bounding box maximum x in millipoints (exclusive) R4 = bounding box maximum y in millipoints (exclusive)
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	<p>This call measures the size of a string without actually printing it. The string can consist of printable characters and all the usual control sequences. The bounds are given relative to the start point of the string (they might be negative due to backward move control sequences, etc).</p> <p>Note that this command cannot be used to measure the screen size of a string because of rounding errors. The string must be scanned 'manually', by stepping along in millipoints, and using <code>Font_ConverttoOS</code> and <code>Font_CharBBox</code> to measure the precise position of each character on the screen. Usually this can be avoided, since text is formatted in rows, which are assumed to be high enough for it.</p>
Related SWIs	<code>Font_ReadInfo</code> (SWI &40084), <code>Font_CharBBox</code> (SWI &4008E)
Related vectors	None

Font_ReadColourTable (SWI &40098)

Read the anti-alias colour table

On entry

R1 = pointer to 16 byte area of memory

On exit

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 returns the 16 entry colour table to the block pointed to by R1 on entry. This contains the 16 colours used by the anti-aliasing software when painting text – that is, the values that would be put into screen memory.

Related SWIs

Font_SetPalette (SWI &40093), Font_SetThresholds (SWI &40095),
Font_SetFontColours (SWI &40092)

Related vectors

None

Font_MakeBitmap (SWI &40099)

Make a font bitmap file

On entry

R1 = font handle, or pointer to font name
R2 = x point size * 16
R3 = y point size * 16
R4 = x dots per inch
R5 = y dots per inch
R6 = flags

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call allows a particular size of a font to be pre-stored in the font's directory so that it can be cached more quickly. It is especially useful if subpixel positioning is to be performed, since this takes a long time if done directly from outlines.

The flags have the following meanings:

Bit	Meaning when set
0	construct f9999x9999 (else b9999x9999)
1	do horizontal subpixel positioning
2	do vertical subpixel positioning
3	just delete old file, without replacing it
4 - 31	reserved (must be 0)

Once a font file has been saved, its subpixel scaling will override the setting of FontMax4/5 currently in force (so, for example, if the font file had horizontal subpixel scaling, then when a font of that size is requested, horizontal subpixel scaling will be used even if FontMax4 is set to 0).

Related SWIs

Font_SetFontMax (SWI &4009B)

Related vectors

None

Font_UnCacheFile (SWI &4009A)

Delete cached font information, or recache it

On entry

R1 = pointer to full filename of file to be removed
R2 = recache flag (0 or 1 – see below)

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

If an application such as !FontEd wishes to overwrite font files without confusing the font manager, it should call this SWI to ensure that any cached information about the file is deleted.

The filename pointed to by R1 must be the full filename (ie in the format used by the Filer), and must also correspond to the relevant name as it would have been constructed from Font\$Path and the font name. This means that each of the elements of Font\$Path must be proper full pathnames, including filing system prefix and any required special fields (eg. net:*fileserver:\$.fonts.).

The SWI must be called twice: once to remove the old version of the data, and once to load in the new version. This is especially important in the case of IntMetrics files, since the font cache can get into an inconsistent state if the new data is not read in immediately.

The 'recache' flag in R2 determines whether the new data is to be loaded in or not, and might be used like this:

```
SYS "Font_UnCacheFile",,"<filename>",<replace old file with new one>,0  
SYS "Font_UnCacheFile",,"<filename>",<replace old file with new one>,1
```

Related SWIs

None

Related vectors

None

Font_SetFontMax (SWI &4009B)

Set the FontMax values

On entry

R0 = new value of FontMax (bytes)

R1 - R5 = new values of FontMax1..FontMax5 (pixcls * 72 * 16)

R6, R7 reserved (must be zero)

On exit

—

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call can be used to set the values of FontMax and FontMax1...FontMax5. Changing the configured settings will also change these internal settings, but Font_SetFontMax does not affect the configured values, which come into effect on ctrl-break or when the font manager is re-initialised.

This call also causes the font manager to search through the cache, checking to see if anything would have been cached differently if the new settings had been in force at the time. If so, the relevant data is discarded, and will be reloaded using the new settings when next required.

Related SWIs

Font_ReadFontMax (SWI &4009C)

Related vectors

None

Font_ReadFontMax (SWI &4009C)

Read the FontMax values

On entry

—

On exit

R0 = value of FontMax (bytes)

R1 - R5 = values of FontMax1..FontMax5 (pixels * 72 * 16)

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 can be used to read the values of FontMax and FontMax1...FontMax5. It reads the values that the font manager holds internally (which may have been altered from the configured values by Font_SetFontMax).

Related SWIs

Font_SetFontMax (SWI &4009B)

Related vectors

None

Font_ReadFontPrefix (SWI &4009D)

Find the directory prefix for a given font handle

On entry

R0 = font handle
R1 = pointer to buffer
R2 = length of buffer

On exit

R1 = pointer to terminating null
R2 = bytes remaining in buffer

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 finds the directory prefix relating to a given font handle, which indicates where the font's IntMetrics file is, and copies it into the buffer pointed to by R1; for example:

```
adfs::4.$.!Fonts.Trinity.Medium.
```

One use for this prefix would be to find out which sizes of a font were available pre-scaled in the font directory.

Related SWIs

None

Related vectors

None

*Commands

*Configure FontMax

Set the maximum font cache size

Syntax

```
*Configure FontMax <n>[K]
```

Parameters

<n> number of 4k chunks
<n>K number of kilobytes

Use

The difference between FontSize and FontMax is the amount of memory that the font manager will attempt to use if it needs to. If other parts of the system have already claimed all the spare memory, then FontSize is what it is forced to work with.

If FontMax is bigger than FontSize, the font manager will attempt to expand the cache if it cannot obtain enough cache memory by throwing away unused blocks (ie. ones that belong to fonts which have had Font_FindFont called on them more often than Font_LoseFont). Once the cache has expanded up to FontMax, the font manager will throw away the oldest block found, even if it is in use. This can result in the font manager heavily using the filing system, since during a window redraw it is possible that all fonts will have to be thrown away and recached in turn.

Example

```
*Configure FontMax 256K
```

Related commands

```
*Configure FontSize
```

Related SWIs

```
Font_CacheAddress   (SWI &40080),   Font_SerFontMax    (SWI &4009B),  
Font_ReadFontMax   (SWI &4009C)
```

Related vectors

None

*Configure FontMax1

Sets the maximum height at which to use a non-exact font from an x90y45 file

Syntax

```
*Configure FontMax1 <max height>
```

Parameters

<max height> maximum font pixel height at which to use non-exact font from an x90y45 file

Use

If a font has both an x90y45 file and an Outlines file, the font manager is in something of a quandary. It would rather use the outlines file in all cases, since it always produces results at least as good as the scaled bitmaps, but unfortunately it does take longer.

The solution is that the font manager will use the x90y45 version of a font either if the exact size required is contained in the file, or if the font size required is less than or equal to the value specified in FontMax1.

Note that the f9999x9999 (or b9999x9999, as appropriate) will always be preferred if the exact size is found, and it is also possible to scale from an f9999x9999 file, by creating an x90y45 file which contains only the name of the f9999x9999 file. In the latter case, the same rules apply concerning FontMax1, if there is also an Outlines file.

The height is set in pixels rather than points because it is the pixel size that affects cache usage. This corresponds to different point sizes on different resolution output devices:

$$\text{pixel height} = \text{height in points} * \text{pixels (or dots) per inch} / 72$$

Example

```
*Configure FontMax1 25
```

Related commands

```
*Configure FontMax2
```

Related SWIs

```
Font_SetFontMax (SWI &4009B), Font_ReadFontMax (SWI &4009C)
```

Related vectors

None

*Configure FontMax2

Sets the maximum height at which to use anti-aliased outlines

Syntax

```
*Configure FontMax2 <max height>
```

Parameters

<max height> maximum font pixel height at which to use anti-aliased outlines

Use

If the font size required is larger than <max height>, then the font manager will never convert from outlines to anti-aliased (4-bit-per-pixel) bitmaps, but will use 1-bit-per-pixel bitmaps instead. These only use a quarter of the cache space that anti-aliased bitmaps would, and are quicker to convert from outlines.

The font manager will use an f9999x9999 or x90y45 version of the font if the exact size is found or the font size is less than or equal to FontMax1.

The height is set in pixels rather than points because it is the pixel size that affects cache usage. This corresponds to different point sizes on different resolution output devices:

$$\text{pixel height} = \text{height in points} * \text{pixels (or dots) per inch} / 72$$

Example

```
*Configure FontMax2 20
```

Related commands

```
*Configure FontMax1
```

Related SWIs

```
Font_SetFontMax (SWI &4009B), Font_ReadFontMax (SWI &4009C)
```

Related vectors

None

*Configure FontMax3

Sets the maximum height at which to retain bitmaps in the cache

Syntax

```
*Configure FontMax3 <max height>
```

Parameters

<max height> maximum font pixel height at which to retain bitmaps in the cache

Use

If the font size required is larger than <max height>, the font manager will not store the results of converting from outlines to bitmaps, but will instead draw the data directly onto the destination, cacheing the outlines themselves instead. Note that in this case the text is not drawn anti-aliased, since the Draw module is used to draw the outlines directly.

The font manager sets up the appropriate GCOL and TINT settings when drawing the outlines, but it resets them afterwards.

The height is set in pixels rather than points because it is the pixel size that affects cache usage. This corresponds to different point sizes on different resolution output devices:

$$\text{pixel height} = \text{height in points} * \text{pixels (or dots) per inch} / 72$$

Example

```
*Configure FontMax3 35
```

Related commands

None

Related SWIs

Font_SetFontMax (SWI &4009B), Font_ReadFontMax (SWI &4009C)

Related vectors

None

*Configure FontMax4

Sets the maximum width at which to use horizontal subpixel anti-aliasing

Syntax	<code>*Configure FontMax4 <max width></code>
Parameters	<code><max width></code> maximum font pixel width at which to use horizontal subpixel anti-aliasing
Use	<p>If the font width is less than or equal to <code><max width></code>, the font manager will try to use the outlines file rather than <code>x90y45</code>, and will construct 4 bitmaps for each character, corresponding to 4 possible horizontal subpixel alignments on the screen. When painting the text, it will use the bitmap which corresponds most closely to the required alignment.</p> <p>Note that if there is an <code>f9999x9999</code> file of the appropriate size, this will take precedence over the settings of <code>FontMax4</code> and <code>FontMax5</code>. This bitmap may however have been constructed with subpixel positioning already performed (see <code>Font_MakeBitmap</code>).</p> <p>The width is set in pixels rather than points because it is the pixel size that affects cache usage. This corresponds to different point sizes on different resolution output devices:</p> $\text{pixel width} = \text{width in points} * \text{pixels (or dots) per inch} / 72$
Example	<code>*Configure FontMax4 0</code>
Related commands	<code>*Configure FontMax5</code>
Related SWIs	<code>Font_SetFontMax (SWI &4009B)</code> , <code>Font_ReadFontMax (SWI &4009C)</code>
Related vectors	None

*Configure FontMax5

Sets the maximum height at which to use vertical subpixel anti-aliasing

Syntax

```
*Configure FontMax5 <max height>
```

Parameters

<max height> maximum font pixel height at which to use vertical subpixel anti-aliasing

Use

If the font height is less than or equal to <max height>, the font manager will try to use the outlines file rather than x90y45, and will construct 4 bitmaps for each character, corresponding to 4 possible vertical subpixel alignments on the screen. When painting the text, it will use the bitmap which corresponds most closely to the required alignment.

Note that if there is an f9999x9999 file of the appropriate size, this will take precedence over the settings of FontMax4 and FontMax5. This bitmap may however have been constructed with subpixel positioning already performed (see Font_MakeBitmap).

The height is set in pixels rather than points because it is the pixel size that affects cache usage. This corresponds to different point sizes on different resolution output devices:

$$\text{pixel height} = \text{height in points} * \text{pixels (or dots) per inch} / 72$$

Example

```
*Configure FontMax5 0
```

Related commands

```
*Configure FontMax4
```

Related SWIs

```
Font_SetFontMax (SWI &4009B), Font_ReadFontMax (SWI &4009C)
```

Related vectors

None

*Configure FontSize

Set the amount of space that is initially given to the font manager

Syntax

```
*Configure FontSize <n>K
```

Parameters

<n> number of kilobytes to allocate

Use

FontSize refers to the initial cache size, which is set when the font manager is first initialised. The minimum cache size can also be changed from the Task Manager, by dragging the font cache bar directly, although this is not remembered when after a Control-reset.

Example

```
*Configure FontSize 32K
```

Related commands

```
*Configure FontMax
```

Related SWIs

```
Font_CacheAddress (SWI &40080)
```

Related vectors

None

*FontCat

	List the available fonts
Syntax	*FontCat [<pathname>]
Parameters	<pathname> optional pathname
Use	This command searches through the list of font directories and lists the fonts that are available. If no pathname is specified, the system variable Font\$Path is used. Font_FindFont uses the same variable when it searches for a font.
Example	*FontCat adfs:\$.Fonts. The last '.' is essential Corpus.Medium Portrhouse.Standard Trinity.Medium
Related commands	None
Related SWIs	Font_FindFont (SWI &40081), Font_ListFonts (SWI &40091)
Related vectors	None

*FontList

Displays the fonts and free space in the font cache

Syntax

```
*FontList
```

Parameters

—

Use

*FontList displays the fonts currently in the font cache. For each font, details are given of its point size, its resolution, the number of times it is being used by various applications, and the amount of memory it is using. The size of the font cache and the amount of free space is also given.

Example

```
*FontList
      Name           Size           Dots/inch       Use       Memory
      ----           -
1.Trinity.Medium    12 point        90x45           0         3 Kbytes
2.Corpus.Standard   14 point        90x45           2         11 Kbytes

Cache size: 24 Kbytes
free: 9 Kbytes
```

Related commands

None

Related SWIs

Font_ListFonts

Related vectors

None

Application Notes

BASIC example of justified text

```
100 SYS "Font_FindFont",,"Trinity.Medium",320,320,0,0 TO HAN%
110 REM sets font handle
120 SYS "Font_SetPalette",,8,9,6,&FFFFFF00,&00000000
130 REM Set the palette to use colours 8-15 as white to black
140 MOVE 800,500
150 REM Set the right hand side of justification
160 SYS "Font_Paint",,"This is a test",&11,0,500
170 SYS "Font_LoseFont",HAN%
```

On line 160, Font_Paint is being told to use OS coordinates and justify, starting at location 0,500. 800,500 has been declared as the right hand side of justification by line 140.



Draw module

Introduction

The Draw module is an implementation of PostScript type drawing. A collection of moves, lines, and curves in a user-defined coordinate system are grouped together and can be manipulated as one object, called a path.

A path can be manipulated in memory or upon writing to the VDU. There is full control over the following characteristics of it:

- rotation, scaling and translation of the path
- thickness of a line
- description of dots and dashes for a line
- joins between lines can be mitred, round or bevelled
- the leading or trailing end of a line, or dot (which are in fact just very short dashes), can be butt, round, a projecting square or triangular (used for arrows)
- filling of arbitrary shapes
- what the fill considers to be interior

A path can be displayed in many different ways. For example, if you write a path that draws a petal, and draw it several times rotating about a point, you will have a flower. This uses only one of the characteristics that you can control.

The Draw application was written using this module, and this is the kind of application that it is suited to. It is advisable to read the section on Draw in the User Guide to familiarise yourself with some of the properties of the Draw module.

Overview

There are many specialised terms used within the Draw module. Here are the most important ones. If you are familiar with PostScript, then many of these should be the same.

- A *path element* is a sequence of words. The first word in the sequence has a command number, called the *element type*, in the bottom byte. Following this are parameters for that element type.
- A *subpath* is a sequence of path elements that defines a single connected polygon or curve. The ends of the subpath may be connected, so it forms a loop (in which case it is said to be *closed*) or may be *loose ends* (in which case it is said to be *open*). A subpath can cross itself or other subpaths in the same path.

See below for a more detailed explanation of when a subpath is open or closed.

- A *path* is a sequence of subpaths and path elements.
- A *Bezier curve* is a type of smooth curve connecting two *endpoints*, with its direction and curvature controlled by two *control points*.
- *Flattening* is the process of converting a Bezier curve into a series of small lines when outputting.
- *Flatness* is how closely the lines will approximate the original Bezier curve.
- A *transformation matrix* is the standard mathematical tool for two-dimensional transformations using a three by three array. It can rotate, scale and translate (move).
- To *stroke* means to draw a thickened line centred on a path.
- A *gap* is effectively a transparent line segment in a subpath. If the subpath is stroked, the piece around the gap will not be plotted. Gaps are used by Draw to implement dashed lines.
- *Line caps* are placed at the ends of an open subpath and at the ends of dashes in a dashed line when they are stroked. They can be *butt*, *round*, a *projecting square* or *triangular*.
- *Joins* occur between adjacent lines, and between the start and end of a closed subpath. They can be *mitred*, *round* or *bevelled*.

- To *Fill* means to draw everything inside a path.
- *Interior* pixels are ones that are filled. *Exterior* pixels are not filled.
- A *winding number rule* is the rule for deciding what is interior or exterior to a path when filling. The interior parts are those that are filled.
- *Boundary* pixels are those that would be drawn if the line were stroked with minimum thickness for the VDU.
- *Thickening* a path is converting it to the required thickness – that is generating a path which, if filled, would produce the same results as stroking the original path.

Scaling systems

This is an area where you must take great care when using the Draw module, because four different systems are used in different places.

OS units

OS units are notionally 1/180th of an inch, and are the standard units used by the VDU drivers for specifying output to the screen

This coordinate system is (not surprisingly) what the Draw module uses when it strokes a path onto the screen.

Internal Draw units

Internally, Draw uses a coordinate system the units of which are 1/256th of an OS unit. We shall call these internal Draw units.

In a 32 bit internal Draw number, the top 24 bits are the number of OS units, and the bottom 8 bits are the fraction of an OS unit.8 fixed point system.

User units

The coordinates used in a path can be in any units that you wish to use. These are translated by the transformation matrix into internal Draw units when generating output.

Note that because it is a fixed point system, scaling problems can occur if the range is too far from the internal Draw units. Because of this problem, you are limited in the range of user units that you can use.

Transform units

Transform units are only used to specify some numbers in the transformation matrix. They divide a word into two parts: the top two bytes are the integer part, and the bottom two bytes are the fraction part.

Transformation matrix

This is a three by three matrix that can be used to rotate, scale or translate a path in a single operation. It is laid out like this:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

This matrix transforms a coordinate (x,y) into another coordinate (x',y') as follows:

$$\begin{aligned} x' &= ax + cy + e \\ y' &= bx + dy + f \end{aligned}$$

The common transformations can all be easily done with this matrix. Translation by a given displacement is done by e for the x axis and f for the y axis. Scaling the x axis uses a , while the y axis uses d . Rotation can be performed by setting $a=\cos(\theta)$, $b=\sin(\theta)$, $c=-\sin(\theta)$ and $d=\cos(\theta)$, where θ is the angle of rotation.

a , b , c and d are given in transform units to allow accurate specification of the fractional part. e and f are specified in internal Draw units, so that the integer part can be large enough to adequately specify displacements on the screen. (Were transform units to be used for these coefficients, then the maximum displacement would only be 256 OS units, which is not very far on the screen.)

Winding rules

The winding rule determines what the Draw module considers to be interior, and hence filled.

Even-odd roughly means that an area is filled if it is enclosed by an even number of subpaths. The effect of this is that you will never have two adjacent areas of the same state, ie filled or unfilled.

Non-zero winding fills areas on the basis of the direction in which the subpaths which surround the area were constructed. If an equal number of subpaths in each direction surround the area, it is not filled, otherwise it is.

The positive winding rule will fill an area if it is surrounded by more anti-clockwise subpaths than clockwise. The negative winding rule works in reverse to this.

Even-odd and non-zero winding are printer driver compatible, whereas the other two are not. If you wish to use the path with a printer driver, then bear this in mind.

Stroking and filling

Flattening means bisecting any Bezier curves recursively until each of the resulting small lines lies within a specified distance of the curve. This distance is called flatness. The longer this distance, the more obvious will be the straight lines that approximate the curve.

All moving and drawing is relative to the VDU graphics origin (as set by VDU 29,x;y;).

None of the Draw SWIs will plot outside the boundaries of the VDU graphics window (as set by VDU 24,l;b;r;t;).

All calls use the colour (both pixel pattern and operation) set up for the VDU driver. Note that not all such colours are compatible with printer drivers.

Printing

If your program needs to generate printer output, then it is very important that you read the chapter entitled *Printer Drivers*. The Draw SWIs that are affected by printing have comments in them about the limitations and effects.

Floating point

SWI numbers and names have been allocated to support floating point Draw operations. In fact for every SWI described in this chapter, there is an equivalent one for floating point – just add FP to the end of each name.

The floating point numbers used in the specification are IEEE single precision floating point numbers.

They may be supported in some future version of RISC OS, but if you try to use them in current versions you'll get an error back..

Technical Details

Data structures

Many common structures are used by Draw module SWIs. Rather than duplicate the descriptions of these in each SWI, they are given here. Some SWIs have small variations which are described with the SWI.

Path

The path structure is a sequence of subpaths, each of which is a sequence of elements. Each element is from one to seven words in length. The lower byte of the first word is the element type. The remaining three bytes of it are free for client use. On output to the input path the Draw module will leave these bytes unchanged. However, on output to a standard output path the Draw module will store zeroes in these three bytes.

The element type is a number from 0 to 8 that is followed by the parameters for the element, each a word long. The path elements are as follows:

Element Type	Parameters	Description
0	n	End of path. n is ignored when reading the path, but is used to check space when reading and writing a path.
1	ptr	Pointer to continuation of path. ptr is the address of the first path element of the continuation.
2	x y	Move to (x,y) starting new subpath. The new subpath does affect winding numbers and so is filled normally. This is the normal way to start a new subpath.
3	x y	Move to (x,y) starting new subpath. The new subpath does not affect winding numbers when filling. This is mainly for internal use and rarely used by applications.
4		Close current subpath with a gap.
5		Close current subpath with a line. It is better to use one of these two to close a subpath than 2 or 3, because this guarantees a closed subpath.

6	x1 y1 x2 y2 x3 y3	Bezier curve to (x3,y3) with control points at (x1,y1) and (x2,y2).
7	x y	Gap to (x,y). Do not start a new subpath. Mainly for internal use in dot-dash sequences.
8	x y	Line to (x,y).

You will notice that there are some order constraints on these element types:

- path elements 2 and 3 start new subpaths
- path elements 6, 7 and 8 may only appear while there is a current subpath
- path elements 4 and 5 may only appear while there is a current subpath, and end it, leaving no current subpath
- path elements 2 and 3 can also be used to close the current subpath (which is a part of starting a new subpath).

Open and closed subpaths

When you are stroking (using `OS_DrawStroke`), if a subpath ends with a 4 or 5 then it is closed, and the ends are joined – whereas a 2 or 3 leaves a subpath open, and the loose ends are capped. These four path elements explicitly leave a stroked subpath either open or closed.

Some other operations implicitly close open subpaths, and this will be stated in their descriptions.

Just because the ends of a subpath have the same coordinates, that doesn't mean the subpath is closed. There is no reason why the loose ends of an open subpath cannot be coincident.

Output path

After a SWI has written to an output path, it is identical to an input path. When it is first passed to the SWI as a parameter, the start of the block pointed to should contain an element type zero (end of path) followed by the number of available bytes. This is so that the Draw module will not accidentally overrun the buffer.

Fill style

The fill style is a word that is passed in a call to `Draw_Fill`, `Draw_Stroke`, `Draw_StrokePath` or `Draw_ProcessPath`. It is a bitfield, and all of the calls use at least the following common states. See the description of each call for differences from this:

Bit(s)	Value	Meaning
0, 1	0	non-zero winding number rule.
	1	negative winding number rule.
	2	even-odd winding number rule.
	3	positive winding number rule.
2	0	don't plot non-boundary exterior pixels.
	1	plot non-boundary exterior pixels.
3	0	don't plot boundary exterior pixels.
	1	plot boundary exterior pixels.
4	0	don't plot boundary interior pixels.
	1	plot boundary interior pixels.
5	0	don't plot non-boundary interior pixels.
	1	plot non-boundary interior pixels.
6 - 31	1	reserved – must be written as zero

Matrix

The matrix is passed as pointer to a six word block, in the order a, b, c, d, e, and f as described earlier. That is:

Offset	Value	Common use(s)
0	a	x scale factor, or $\cos(\theta)$ to rotate
4	b	$\sin(\theta)$ to rotate
8	c	$-\sin(\theta)$ to rotate
12	d	y scale factor, or $\cos(\theta)$ to rotate
16	e	x translation
20	f	y translation

If the pointer is zero, then the identity matrix is assumed – no transformation takes place.

Remember that a - d are in Transform units, while e and f are in internal Draw units.

Flatness

Flatness is the maximum distance that a line is allowed to be from a Bezier curve when flattening it. It is expressed in user units. So a smaller flatness will result in a more accurate rendering of the curve, but take more time and space. For very small values of flatness, it is possible to cause the 'No room in RMA' error.

A recommended range for flatness is between half and one pixel. Any less than this and you're wasting time; any more than this and the curve becomes noticeably jagged. A good starting point is:

$$\text{flatness} = \text{number of user units in x axis} / \text{number of pixels in x axis}$$

A value of zero will use the default flatness. This is set to a useful value that balances speed and accuracy when stroking to the VDU using the default scaling.

Note that if you are going to send a path to a high resolution printer, then you may have to set a smaller flatness to avoid jagged curves.

Line thickness

The line thickness is in user coordinates.

- If the thickness is zero then the line is drawn with the minimum width that can be used, given the limitations of the pixel size (so lines are a single pixel wide).
- If the thickness is 2, then the line will be drawn with a thickness of 1 user coordinate translated to pixels on either side of the theoretical line position.
- If the line thickness is non-zero, then the cap and join parameter must also be passed.

Cap and join

The cap and join styles are passed as pointer to a four word block. A pointer of zero can be passed if cap and join are ignored (as they are for zero thickness lines). The block is structured as follows:

Word	Byte	Description
0	0	join style 0 = mitred joins 1 = round joins 2 = bevelled joins
	1	leading cap style 0 = butt caps 1 = round caps 2 = projecting square caps 3 = triangular caps
	2	trailing cap style (as leading cap style)
	3	reserved – must be written as zero.
4		This value must be set if using mitred joins.
	0,1	fractional part of mitre limit for mitre joins
	2,3	integer part of mitre limit for mitre joins
8	0,1	setting for leading triangular cap width on each side. (in 256ths of line widths, so 0100 is 1 linewidth)
	2,3	setting for leading triangular cap length away from the line, in the same measurements as above
12	all	This sets the trailing triangular cap size, using the same structure as the previous word.

The mitre limit is a little more complex than the others, so it is explained here rather than above. At any given corner, the mitre length is the distance from the point at which the inner edges of the stroke meet, to the point where the outer edges of the stroke meet. This distance increases as the angle between the lines decreases. If the ratio of the mitre length to the line width exceeds the mitre limit, stroke treats the corner with a bevel join instead of a mitre join. Also see the notes on scaling, later in this section.

Note that words at offsets 4, 8, and 12 are only used if the appropriate style is selected by the earlier parts. The structure can therefore be made shorter if triangular caps and mitres are not used.

Dash pattern

The dash pattern is passed as a pointer to a block, the size of which is defined at the start, as follows:

Word	Description
0	distance into dash pattern to start in user coordinates
4	number of elements (N) in the dash pattern
8 to 4N+4	elements in the dash pattern, each of which is a distance in user coordinates.

Again the pointer can be zero, which implies that continuous lines are drawn.

Each element specifies a distance to draw in the present state. The pattern starts with the draw on, and alternates off and on for each successive element. If it reaches the end of the pattern while drawing the line, then it will restart at the beginning.

If N is odd, then the elements will alternate on or off with each pass through the pattern. ie. the first element will be on the first pass, off the second pass, on the third pass, and so on.

Scaling

The Draw module uses fixed point arithmetic for speed. The number representations used are chosen to keep rounding errors small enough not to be noticeable.

However, if you use the transformation matrix to scale a path up a great deal, you will also scale up the rounding errors and make them visible.

To avoid such problems, we recommend that you don't use scale factors of more than 8 when converting from User units to internal Draw units. (This maximum recommended scale factor of 8 is 80000 in the Transform units used in the transformation matrix.)

Draw SWIs

Though there are a number of SWIs, they all call Draw_ProcessPath. Because this takes so many parameters, the other SWIs are provided as an easy way of using its functionality.

There are two that output to the VDU. Draw_Stroke emulates the PostScript stroke function and will draw a path onto the VDU. Draw_Fill acts like the fill function and fills the inside of a path. It is likely that most applications will only use these two SWIs.

The others are shortcuts for processing a path in one way or other. `Draw_StrokePath` acts exactly like `Draw_Stroke`, except it puts its output into a path rather than onto the VDU. Filling its output path produces the same results as stroking its input path. `Draw_FlattenPath` will handle only the flattening of a path, writing its output to a path. Likewise, `Draw_TransformPath` will only use the matrix on a path. All these processing SWIs are useful when a path will be sent to the VDU many times. If the path is flattened or transformed before the stroking, then it will be done faster.

Printer drivers

If you are using a printer driver, you should note that it cannot deal with all calls to the Draw module. For full details of this, see the chapter entitled *Printer Drivers*. As a general rule, you should avoid the following features:

- AND, OR, etc operations on colours when writing to the screen.
- Choice of fill style: eg fill excluding/including boundary, fill exterior, etc.
- Positive and negative winding number rules.
- Line cap enhancements, particularly differing leading and trailing caps and triangular caps.

The printer driver will also intercept `DrawV` and modify how parts of the Draw module work. Here is a list of the effects that are common to all the SWIs that output to the VDU normally:

- cannot deal with positive or negative winding numbers
- cannot fill:
 - 1 non-boundary exterior pixels
 - 2 exterior boundary pixels only
 - 3 interior boundary pixels only
 - 4 exterior boundary and interior non-boundary pixels
- an application should not rely on any difference between the following fill states:
 - 1 interior non-boundary pixels only
 - 2 all interior pixels
 - 3 all interior pixels and exterior boundary pixels

SWI Calls

Draw_ProcessPath (SWI &40700)

Main Draw SWI

On entry

R0 = pointer to input path buffer (see below)
R1 = fill style
R2 = pointer to transformation matrix, or 0 for identity matrix
R3 = flatness, or 0 for default
R4 = line thickness, or 0 for default
R5 = pointer to line cap and join specification
R6 = pointer to dash pattern, or 0 for no dashes
R7 = pointer to output path buffer, or value (see below)

On exit

R0 depends on entry value of R7
 if R7 = 0, 1 or 2 R0 is corrupted
 if R7 = 3 R0 = size of output buffer
 if R7 is a pointer R0 = pointer to new end of path indicator
R1 - R7 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

All the other SWIs in the Draw module are translated into calls to this SWI. They are provided to ensure that suitable names exist for common operations and to reduce the number of registers to set up when calling.

The input path, matrix, flatness, line thickness, cap and join, and dash pattern are as specified in the *Technical Details*..

The fill style is as in the *Technical Details*, with the following additions:

Bit(s)	Meaning
6 - 26	reserved – must be written as zero
27	set if open subpaths are to be closed
28	set if the path is to be flattened
29	set if the path is to be thickened
30	set if the path is to be re-flattened after thickening
31	set for floating point output (not implemented)

Normally, the output path will act as described in the *Technical Details*, but with the following changes if the following values are passed in R7:

Value	Meaning
0	Output to the input path buffer. Only valid if the input path's length does not change during the call.
1	Fill the path normally.
2	Fill the path, subpath by subpath. (Draw_Stroke will often use this to economise on RMA usage).
3	Count how large an output buffer is required for the given path and actions.
&80000000+pointer	Output the path's bounding box, in transformed coordinates. The buffer will contain the four words: lowx, lowy, highx, highy.
pointer	Output to a specified output buffer. The length of the buffer must be indicated by putting a suitable path element 0 at the start of the buffer, and a pointer to the new path element 0 is returned in R0 to allow you to append to the output path.

You may do the following things with this call, in this order:

- 1 Open subpaths may be closed (if selected by bit 27 of R1).
- 2 The path may be flattened (if selected by bit 28 of R1). This uses R3.
- 3 The path may be dashed (if $R6 \neq 0$).
- 4 The path may be thickened (if selected by bit 29 of R1). This uses R4 and R5.
- 5 The path may be re-flattened (if selected by bit 30 of R1). This uses R3.
- 6 The path may be transformed (if $R2 \neq 0$).
- 7 Finally, the path is output in one of a number of ways, depending on R7.

Note that R3, R4 and R5 may be left unspecified if the options that use them are not specified.

If you try to dashing, thickening or filling on an unflattened Bezier curve, it will produce an error, as this is not allowed.

If you are using the printer driver, then it will intercept this SWI and affect its operation. In addition to the general comments in the *Technical Details*, it is unable to handle $R7 = 1$ or 2 .

Related SWIs

None

Related vectors

DrawV

Draw_Fill (SWI &40702)

	Process a path and send to VDU, filling the interior portion
On entry	R0 = pointer to input path R1 = fill style, or 0 for default R2 = pointer to transformation matrix, or 0 for identity matrix R3 = flatness, or 0 for default
On exit	R0 corrupted R1 - R3 preserved
Interrupts	Interrupts are enabled Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	SWI is not re-entrant
Use	<p>This command emulates the PostScript 'fill' operator. It performs the following actions:</p> <ul style="list-style-type: none">• closes open subpaths• flattens the path• transforms it to standard coordinates• fills the resulting path and draws to the VDU. <p>The input path, matrix, and flatness are as specified in the <i>Technical Details</i>.</p> <p>The fill style is as specified with the following addition. A fill style of zero is a special case. It specifies a useful default fill style, namely &30. This means fill to halfway through boundary, non-zero rule.</p> <p>If you are using the printer driver, then it will intercept this SWI and affect its operation. See the general comments in the <i>Technical Details</i> section.</p>
Related SWIs	None
Related vectors	DrawV

Draw_Stroke (SWI &40704)

	Process a path and send to VDU
On entry	R0 = pointer to input path R1 = fill style, or 0 for default R2 = pointer to transformation matrix, or 0 for identity matrix R3 = flatness, or 0 for default R4 = line thickness, or 0 for default R5 = pointer to line cap and join specification R6 = pointer to dash pattern, or 0 for no dashes
On exit	R0 corrupted R1 - R6 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 command emulates the PostScript 'stroke' operator. It performs the following actions: <ul style="list-style-type: none">• flattens the path• applies a dash pattern to the path, if R6 \neq 0• thickens the path, using the specified joins and caps• re-flattens the path, to flatten round caps and joins, so that they can be filled.• transforms the path to standard coordinates• fills the resulting path and draws to the VDU

The input path, matrix, flatness, line thickness, cap and join, and dash pattern are as specified in the *Technical Details*.

The fill style is as specified with the following additions. A fill style of zero is a special case. If the line thickness in R4 is non-zero, then it means &30, as in Draw_Fill. If R4 is zero, then &18 is the default, as the flattened and thickened path will have no interior in this case.

If the top bit of the fill style is set, this makes the Draw module plot the stroke all at once rather than one subpath at a time. This means the code will never double plot a pixel, but uses up much more temporary work-space.

If the specified thickness is zero, the added restrictions are that it cannot deal with filling non-boundary exterior pixels and not filling boundary exterior pixels at the same time, ie fill bits 3 - 2 being 01.

If the specified thickness is non-zero, the added restrictions are that it cannot deal with filling just the boundary pixels, ie fill bits 5 - 2 being 0110.

If you are using the printer driver, then it will intercept this SWI and affect its operation. In addition to the general comments in the *Technical Details* section, it has the following effects on fill style.

Most printer drivers will not pay any attention to bit 31 of the fill style - ie plot subpath by subpath or all at once. Use Draw_ProcessPath to get around this problem by processing it before stroking.

Related SWIs

Draw_StrokePath (SWI &40706)

Related vectors

DrawV

Draw_StrokePath (SWI &40706)

Like Draw_Stroke, except writes its output to a path

On entry

R0 = pointer to input path
R1 = pointer to output path, or 0 to calculate output buffer size
R2 = pointer to transformation matrix, or 0 for identity matrix
R3 = flatness, or 0 for default
R4 = line thickness, or 0 for default
R5 = pointer to line cap and join specification
R6 = pointer to dash pattern, or 0 for no dashes

On exit

R0 depends on entry value of R1
 if R1 = 0 R0 = calculated output buffer size
 if R1 = pointer R0 = pointer to end of path marker in output path
R1 - R6 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The input and output paths, matrix, flatness, line thickness, cap and join, and dash pattern are as specified in the *Technical Details*.

This call acts exactly like a call to Draw_Stroke, except that it doesn't write its output to the VDU, but to an output path.

Related SWIs

Draw_Stroke (SWI &40704)

Related vectors

DrawV

Draw_FlattenPath (SWI &40708)

Converts an input path into a flattened output path

On entry

R0 = pointer to input path
R1 = pointer to output path, or 0 to calculate output buffer size
R2 = flatness, or 0 for default

On exit

R0 depends on entry value of R1
 if R1 = 0 R0 = calculated output buffer size
 if R1 = pointer R0 = pointer to end of path marker in output path
R1, R2 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The input and output paths, and flatness are as specified in the *Technical Details*.

This call acts like a subset of Draw_StrokePath. It will only flatten a path. This would be useful if you wanted to stroke a path multiple times and didn't want the speed penalty of flattening the path every time.

Related SWIs

Draw_StrokePath (SWI &40706)

Related vectors

DrawV

Draw_TransformPath (SWI &4070A)

Converts an input path into a transformed output path

On entry

R0 = pointer to input path
R1 = pointer to output path, or 0 to overwrite the input path
R2 = pointer to transformation matrix, or 0 for identity matrix
R3 = 0

On exit

R0 depends on entry value of R1
 if R1 = 0 R0 is corrupted
 if R1 = pointer R0 = pointer to end of path marker in output path
R1 - R3 preserved

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

The input and output paths, and matrix are as specified in the *Technical Details*.

This call acts like a subset of Draw_StrokePath. It will only transform a path. This would be useful if you wanted to stroke a path multiple times and didn't want the speed penalty of transforming the path every time. It is also useful if you want to transform a path before dashing, thickening and so on, to avoid having the rounding errors from the latter operations magnified by the transformation.

Related SWIs

Draw_StrokePath (SWI &40706)

Related vectors

DrawV

Application Notes

Example of simple drawing

The test program that is shown here was devised to represent millimeters internally and scale them to be the correct size when drawn on a particular monitor. Because monitors are different sizes, and even the same model can be adjusted differently in terms of vertical and horizontal picture size, this example would have to be adjusted to suit your particular setup.

This example also has a restriction on screen modes. It will only work on one where the screen is 1280 OS units by 1024 OS units – which most of the current modes are (but not, for example, 132 column modes). This corresponds to 327680 internal Draw units by 262144 internal Draw units.

The first thing to do is to fill the screen with a colour and measure the horizontal and vertical size in millimeters. For this test, the display area measured 210mm across by 160mm down.

Because of scaling limitations, we will work with a user scale of thousandths of millimeters. Thus, there are 210000 user units across and 160000 user units down.

The BASIC program described here is presented in a jumbled order so that the features are described and written one at a time. Once it is all typed in, then it will seem a lot more obvious.

Transformation matrix

The next step is to work out the scaling factors for the transformation matrix. Taking the horizontal size first, we start with 327680 internal Draw units = 210000 user units., giving 1.5604 internal Draw units per user unit. Vertically, 262144 internal Draw units = 160000 user units, giving 1.6384 internal Draw units per user unit.

These figures must now be converted to the Transform units used for scaling in the transformation matrix. The 32 bit Transform number is 2^{16} times the actual value, since its fractional part is 16 bits long. So horizontally we want $2^{16} * 1.5604$, which is 102261 (&18F75), and vertically we want $2^{16} * 1.6384$, which is 107374 (&1A36E).

The transformation matrix is initialised as follows:

$$\begin{bmatrix} \&00018F75 & 0 & 0 \\ 0 & \&0001A36E & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This could be calculated automatically, using the following BASIC code, which, whilst not the most efficient, is hopefully the clearest way of representing it:

```
30 xsize = 210000 : ysize = 160000
40 xscale% = (1280 * 256 / xsize) * &010000
50 yscale% = (1024 * 256 / ysize) * &010000
```

After this, `xscale%` would be `&00018F75` and `yscale%` would be `&0001A36E`, the values to place in the matrix. The matrix would be programmed as follows:

```
20 DIM transform% 23
60 transform%!0 = xscale%           :REM element a in the matrix
70 transform%!4 = 0                 :REM element b
80 transform%!8 = 0                 :REM element c
90 transform%!12 = yscale%         :REM element d
100 transform%!16 = 0               :REM element e
110 transform%!20 = 0               :REM element f
```

Important

It is important to remember that, whilst this example is using thousandths of millimeters as its internal coordinate system, they could be anything within the valid limits. Draw is not affected by what they are. Using the technique described above, ANY valid units can be used. We used 210000 by 160000 user units for our scale; it could be 500000 by 350000 or 654363 by 314159 or whatever. This program will work with all valid scales, simply by changing the definitions of `xsize` and `ysize`.

Creating the path

In order to create the path, this simple program uses a procedure to put a single word into the path and advance the pointer. In a large application, it would be a good idea to write individual routines to generate each element type, because this technique would become tedious in a large program.

This preamble defines what needs to be at the start of the program. Notice that line 20 overwrites the earlier definition.

```
10 pathlength% = 256
20 DIM path% pathlength% - 1, transform% 23
160 pathptr% = 0           :REM Initialise the pointer
```

Later on in the program would be the procedure to add a word to the path

```
320 END
330 DEF PROCadd(value%)
340 IF pathptr%+4 > pathlength% THEN ERROR 0,"Insufficient path buffer"
350 path%!pathptr% = value%
360 pathptr% += 4
370 ENDPROC
```

The simple path shown here generates a rectangle with no bottom line. It is 90mm by 40mm and offset by 80mm in the x and y axes from the origin.

```
170 PROCadd(2) : PROCadd(80000) : PROCadd(80000) :REM Move to start
180 PROCadd(8) : PROCadd(80000) : PROCadd(120000) :REM Draw
190 PROCadd(8) : PROCadd(170000) : PROCadd(120000)
200 PROCadd(8) : PROCadd(170000) : PROCadd(80000)
250 PROCadd(4) : REM Close the subpath. PROCadd(5) would close the rectangle
260 PROCadd(0) : PROCadd(pathlength%-pathptr%-4) :REM End path
```

Simple stroke

Once the path and the transformation matrix have been completed, all that remains is to set the graphics origin and stroke the path onto the screen.

```
270 VDU 29,0;0;
280 SYS "Draw_Stroke",path%,0,transform%,0,0,0,0
```

Translation

Another matrix operation that can be performed is translation, or moving. Remember that the parameters in the matrix are in internal Draw coordinates, not the millimeters used in this example as user coordinates. If you want to translate in OS coordinates, then the translation must be multiplied by 256.

In this example, we are going to re-stroke the path, translated 60 OS units in x and -100 OS units in y.

```
290 transform%!16 = 60<<8
300 transform%!20 = -100<<8
310 SYS "Draw_Stroke",path%,0,transform%,0,0,0,0
```

You will now see two versions of the path, the new one 100 OS units lower and 60 OS units shifted to the right.

Similarly, the matrix may be modified to rotate the path. If you aren't sure how to do this, then see any mathematical text on matrix arithmetic.

Curves

In order to add a curve to the path, we will add a new subpath to the section that creates the path. This curve draws an alpha shape. Note that element type 2 implicitly closes the initial subpath:

```
210 PROCadd(2) : PROCadd(50000) : PROCadd(50000) :REM x1,y1
220 PROCadd(6) : PROCadd(80000) : PROCadd(80000) :REM x2,y2
230 PROCadd(85000) : PROCadd(30000) :REM x3,y3
240 PROCadd(50000) : PROCadd(60000) :REM x4,y4
```

Whilst the flatness can be left at its default value, this shows how the stroke commands can be changed to set the flatness to a sensible value. 640 is used because this program was run in a 640 pixel mode.

```
280 SYS "Draw_Stroke",path%,0,transform%,xsize/640,0,0,0
310 SYS "Draw_Stroke",path%,0,transform%,xsize/640,0,0,0
```

Line thickness

To make the lines shown thicker than the default, it is necessary to specify a thickness and also the joins and caps block. Notice that line 20 has been changed to allocate space for the joins and caps block. We will use round caps and bevelled joints.

```
20 DIM path% pathlength%-1, transform% 23, joinsandcaps% 15
120 joinsandcaps%!0 = 4010102
130 joinsandcaps%!4 = 0
140 joinsandcaps%!8 = 0
150 joinsandcaps%!12 = 0
```

Now all that remains is to change the stroke commands to specify a thickness and point to the block just specified. For this example we will make the first stroke 5000 units (5mm) thick and the second one half that:

```
280 SYS "Draw_Stroke",path%,0,transform%,xsize/640, 5000,joinsandcaps%,0
310 SYS "Draw_Stroke",path%,0,transform%,xsize/640, 2500,joinsandcaps%,0
```

Plainly, there are many more features that could be added to this program. But you should have the idea now of how it fits together and be able to experiment for yourself.

Printer Drivers

Introduction

One of the major headaches on some operating systems is that all applications must write drivers for all the required types of printers. This duplicates a lot of work and makes each application correspondingly larger and more complex.

The solution to this problem that RISC OS has adopted is to supply a virtual printer interface, so that all printer devices can be used in the same way. Thus, your application can write to the printer, without being aware of the differences between, for example, a dot matrix or PostScript printer or an XY plotter.

To simplify printer driving further, the printer can be driven with a subset of the same calls that normally write to the screen. Calls to the VDU drivers and to the SpriteExtend, Draw, ColourTrans and Font modules are trapped by the printer driver. It interprets all these calls in the most appropriate way for the selected printer. Where possible, the greater resolution of most printers is used to its fullest advantage.

Of course, not all calls have meaning to the printer driver – flashing colours for example. These generate an error or are ignored as appropriate.

Printer drivers are written to support a general class of printers, such as PostScript printers. They each have a matching desktop application that allows users to control their unique attributes. Thus, applications need not know about printer specific operation, but this does not result in lack of fine control of the printer.

Overview

A printer driver is implemented in RISC OS as a relocatable module. It supplies SWIs concerned with starting, stopping and controlling a print job.

Rectangles

A key feature of all printer drivers is the rectangle. In normal use, it is a page. It is however possible to have many rectangles appear on the same physical sheet of paper. For example, an A3 sized plotter may be used to draw two A4 rectangles on it side by side; or it could be used to generate a pagination sheet for a DTP package, showing many rectangles on a sheet.

When reading this chapter, in most cases you can consider a rectangle and a page to be effectively equivalent, but bear in mind the above use of rectangles.

Measurement systems

Many of the printer driver SWIs deal with an internal measurement system, using millipoints. This is 1/1000th of a point, or 1/72000th of an inch. This system is an abstraction from the physical characteristics of the printer. Printed text and graphics can be manipulated by its size, rather than in terms of numbers of print pixels, which will vary from printer to printer.

OS units

OS units are the coordinate system normally used by the VDU drivers. In this context, an OS unit is defined as 1/180th of an inch, so each OS unit is 2/5ths of a point, or 400 millipoints.

It is in this coordinate system that all plotting commands are interpreted. When a rectangle is declared, it is given a size in OS units. This is treated like a graphics window, with output outside it being clipped, and so on.

Transform matrix

Like the Draw module, the printer driver uses a transform matrix to convert OS units to the scale, rotation and translation required on paper. With a matrix with no scaling transformation, a line of 180 OS units, or one inch, will appear as an approximation of an inch long line on all printers. Naturally, it depends on the resolution of the printer as to how close to this it gets. If the matrix scaled x and y up by two, then the line would be two inches long.

Using the printer driver

To send output to the printer, an application must engage in a dialogue with the printer driver. This is similar in part to the dialogue used with the Wimp when a window needs redrawing.

The application starts by opening a file to receive the printer driver's output. The file can be the printer, or a file on any filing system. It passes this to the printer driver to start a print job.

For each page, the application goes through the following steps:

- 1 Pass the printer driver a description of each rectangle to use for the page.
- 2 Tell the printer driver to start drawing the page. It will return with an ID for the first rectangle it needs.
- 3 Go through the printer output using calls to the VDU, Draw, Font, etc.
- 4 Ask the printer for the next rectangle and repeat stage 3
- 5 Repeat stages 3 and 4 as often as required. The printer driver will tell you when it no longer requires any output.

The printer driver will ask either for all of, or for a section of a rectangle you specified. It may ask for a given rectangle once, or many times. A dot matrix driver, for instance, may get the output a strip at a time to conserve workspace, whereas a PostScript driver can send the lot out to the printer in one go.

The point is that you should have no preconceptions about how many times the printer driver will ask for a rectangle, or the order in which it requests rectangles.

When all the required pages have been printed, you issue a SWI to finish the print job and then close the file.

See the example at the end of this chapter for a practical guide to this process.

SWI interception

The printer driver works by trapping all calls to the VDU drivers and to the SpriteExtend, Draw, ColourTrans and Font modules. It will pass some on to the destination module unchanged. Some will generate an error because they cannot be interpreted by the printer driver. Some will be discarded. The ones that are of most interest are taken by the printer driver and interpreted in the

most appropriate way for the printer. The section entitled *Technical Details* in this chapter describes how each module's calls are interpreted by the printer driver.

Technical Details

Printer driver SWIs

Printer information

Though an application shouldn't need to look at all its information, PDriver_Info (SWI &80140) will provide information about the nature of the printer. This includes the:

- type of printer
- x and y resolution
- colour and shading capabilities
- name of the printer (applications usually need to look at this)
- ability to handle filled shapes, thick lines, screen dumps and transformations

PDriver_CheckFeatures (SWI &80142) allows an application to check the printer features described above. This means that an application could change the way it works depending on some general features of the printer.

Much as this system tries to avoid this sort of thing, it is inevitable in some cases. For example, an application that uses lots of sprites on screen will have to go about printing in a different way on an XY plotter. Many colour limitations, however, are solved using halftoning.

PDriver_PageSize (SWI &80143) returns the size of paper and printable area on it. This is used to calculate what size of rectangle to use on it.

Starting a print job

To open a print job, you should first open "printer:" as a file. This device independent name is used because the printer driver application has control over the OS_Byte 5 settings of printer destination (see the chapter entitled *Character output* for details of OS_Byte 5).

You may open any other valid pathname as a file to use as a printer output. The file created may subsequently be dumped to the the printer. This technique could be used for background printing, for instance.

Controlling a print job

The file handle is passed to `PDriver_SelectJob` (SWI &80145). It suspends the current print job, if there is one, and makes the handle you passed the current one. It is the application's responsibility to do this at the right time, because it has sole control over what gets printed at any time on the machine it is running on. Needless to say, a network printer spooler can cope with print commands coming from many machines.

A simple use of the printer driver is to call `PDriver_ScreenDump` (SWI &8014F) which will dump the screen to the printer, if it can handle it. See also the description of screen dumps in the chapter entitled *Sprites*.

`PDriver_CurrentJob` (SWI &80146) will tell you the file handle for the currently active print job.

`PDriver_EnumerateJob` (SWI &80150) allows you to scan through all the print jobs that the printer driver currently knows about.

`PDriver_EndJob` (SWI &80148) will end a job and remove the file handle from the printer driver's internal lists. It will issue all the closing commands to the printer to flush any pages in progress. The file should be closed after doing this, to formally finish the print job.

`PDriver_AbortJob` (SWI &80149) is a more forceful termination. It should be called after any errors while printing. It guarantees that no more commands will be sent to the printer after it.

`PDriver_CancelJob` (SWI &8014E) will cancel a job. It is normally followed by the job being aborted. It is not intended to be used by the printing application, but by another task that allows cancellations of print jobs. It would use `PDriver_EnumerateJobs` to find out which jobs exist and then cancel what it wishes to. The application that owns the cancelled job would subsequently find that it had been cancelled and would then abort the job.

`PDriver_Reset` (SWI &8014A) will abort all print jobs known to the printer driver. Normally, you should never have to use this command. It may be useful during development of an application as an emergency recovery measure.

Printing a page

There are two phases to printing a page. First you must specify all the rectangles to use on the page with `PDriver_GiveRectangle` (SWI &8014B). Each rectangle has a size, transformation matrix, position on the page and rectangle ID specified by you.

Then you call `PDriver_DrawPage` (SWI &8014C) to start the print phase. It returns the first rectangle to output. This may be only a strip of the rectangle you specified, if the printer driver cannot do it all at once. This call is followed by repeated calls to `PDriver_GetRectangle` (SWI &8014D) until it returns saying that there are no more rectangles to print.

The printer driver is free to request rectangles in any order it pleases and as many times as it pleases. For each rectangle request, you must redraw that part of the rectangle.

See the example at the end of this chapter for a practical guide to the sequence to use.

Private SWIs

Some SWIs are used in the interface between the printer driver desktop application, the printer driver, and the font manager. They are briefly described in this chapter, but you must not use them. If nothing else, the interface is not guaranteed because it is a private one. These are the private SWIs:

- `PDriver_SetInfo` (SWI &80141)
- `PDriver_SetPageSize` (SWI &80144)
- `PDriver_FontSWI` (SWI &80146)
- `PDriver_SetPrinter` (SWI &80151)

Trapping of screen SWIs

When a printer driver is running, it intercepts the following vectors:

- `WrchV`
- `SpriteV`
- `DrawV`
- `ColourV`
- `ByteV`

Many of the calls that pass through these vectors will be passed unchanged through the printer driver. However some calls are trapped. In some cases they are changed to something appropriate, and in others generate an error because they cannot be implemented. In addition, the font manager SWIs are trapped through an internal mechanism.

Below, we pass section by section through the effects of the printer driver on the calls that pass through these vectors.

WrchV

Whenever a print job is active, the printer driver will intercept all characters sent through WrchV. It will then queue them in the same way as the VDU drivers do and process complete VDU sequences as they appear. Because the printer driver will not pick up any data currently in the VDU queue, and may send sequences of its own to the VDU drivers, a print job should not be selected with an incomplete sequence in the VDU queue.

OS_Byte 3

Also, because the printer driver may send sequences of its own to the VDU drivers, the output stream specification set by OS_Byte 3 should be in its standard state – as though set by OS_Byte 3,0.

Commands passed on to the VDU

The printer driver will pass the following VDU sequences through to the normal VDU drivers, either because they control the screen hardware or because they affect global resources such as the character and ECF definitions:

VDU 7	Produce bell sound
VDU 19,1,p,r,g,b	Change hardware palette
VDU 20	Set default hardware palette
VDU 23,0,n,m l	Program pseudo-6845 registers
VDU 23,1,n l	Change cursor appearance
VDU 23,2-5,a,b,c,d,e,f,g,h	Set ECF pattern
VDU 23,9-10,n l	Set flash durations
VDU 23,11 l	Set default ECF patterns
VDU 23,12-15,a,b,c,d,e,f,g,h	Simple setting of ECF pattern
VDU 23,17,4,m l	Set ECF type
VDU 23,17,6,x;y; l	Set ECF origin
VDU 23,32-255,a,b,c,d,e,f,g,h	Define character

The printer driver will interpret or fault all other VDU sequences. If the printer driver currently wants a rectangle printed, these will result in it producing appropriate output or errors – that is, if there has been a call to PDriver_DrawPage or PDriver_GetRectangle and the last such call returned $R0 \neq 0$. Otherwise, the printer driver will keep track of some state information – for example, the current foreground and background colours – but will not produce any printer output.

Error commands

The printer driver will always behave as though it is in VDU 5 state. No text coordinate system is defined, and no scrolling is possible. For these reasons, the following VDU sequences are faulted:

VDU 4	exit VDU 5 state
VDU 23,7,m,d,z	scroll display
VDU 23,8,t1,t2,x1,y1,x2,y2	clear text block

VDU printer

It is generally meaningless to try to send or echo characters directly to the printer while printing. Furthermore, attempts to do so are likely to disrupt the operation of printer drivers. For these reasons, the following VDU sequences are faulted:

VDU 1,c	send character to printer
VDU 2	start echoing characters to printer

Screen mode

It is not possible to change the "mode" of a printed page, so the following VDU sequence is faulted:

VDU 22,m	change display mode
----------	---------------------

Reserved calls

A printer driver cannot be written to deal with undefined or reserved calls, so the following VDU sequences are faulted:

VDU 23,18-24,...	reserved for Acorn expansion
VDU 23,28-31,...	reserved for use by applications
VDU 25,216-231,...	reserved for Acorn expansion
VDU 25,240-255,...	reserved for use by applications

Ignored

The following VDU sequences are ignored, either because they normally do nothing (at least when stuck in VDU 5 mode and not echoing characters to the printer) or because they have no sensible interpretation when output to anything other than a screen.

VDU 0	do nothing
VDU 3	stop echoing characters to printer
VDU 5	enter VDU 5 state
VDU 14	start "paged" display
VDU 15	end "paged" display
VDU 17,c	define text colour
VDU 23,17,51	exchange text foreground and background
VDU 27	do nothing
VDU 28,l,b,r,t	define text window

Colours

Colours are a rather complicated matter. It is strongly recommended that applications should use `ColourTrans_SetGCOL`, `ColourTrans_SelectTable` and `ColourTrans_SetFontColours` to set colours, as these will allow the printer to produce as accurate an approximation as it can to the desired colour, independently of the screen palette. The `GCOL` sequence (VDU 18,k,c) should only be used if absolutely necessary, and you should be aware of the fact that the printer driver has a simplified interpretation of the parameters, as follows:

- The fact that the background colour is affected if $c \geq 128$ and the foreground colour if $c < 128$ is unchanged.
- If $k \bmod 8 \neq 0$, subsequent plots and sprite plots will not do anything.
- If $k=0$, subsequent plots will cause the colour $c \bmod 128$ (possibly modified by the current tint) to be looked up in the screen palette at the time of plotting (rather than the time the VDU 18,k,c command was issued). Plotting is done by overwriting with the closest approximation the printer can produce to the RGB combination found. Subsequent sprite plotting will be done without use of the sprite's mask.
- If $k=8$, subsequent plots will be treated the same as $k=0$ above, except that sprite plots will be done using the sprite's mask, if any.
- If $k > 8$, an unspecified solid colour will be used.

VDU 18

The major problems with the use of VDU 18,k,c to set colours are:

- that it makes the printer driver output dependent on the current screen mode and palette.
- that it artificially limits the printer driver to the number of colours displayed on the screen, which can be very limiting in a two colour mode.

Other GCOLs

Other techniques that depend on GCOLs have the same problems and are similarly not recommended: for example `Font_SetFontColours`, colour-changing sequences in strings passed to `Font_Paint`, plotting sprites without a translation table, and so on.

No operations other than overwriting are permitted, mainly because they cannot be implemented on many common printers – such as PostScript printers.

Foreground and background colours

Note that the printer driver maintains its own foreground and background colour information. The screen foreground and background colours are not affected by VDU 18,k,c sequences encountered while a print job is active.

VDU 23,17

Similarly, VDU 23,17,2-3,t| sequences encountered while a print job is active do not affect the screen tints, just the printer driver's own tints. VDU 23,17,0-1,t| sequences would only affect the colours of the text tints, so the printer driver ignores them.

Other graphics state operations

The VDU 6 and VDU 21 sequences have their normal effects of enabling and disabling execution, but not parsing, of subsequent VDU sequences. As usual, the printer driver keeps track of this independently of the VDU drivers.

Cursor movement

The cursor movement VDU sequences (ie. VDU 8-11, VDU 13, VDU 30 and VDU 31,x,y) all update the current graphics position without updating the previous graphics positions, precisely as they do in VDU 5 mode on the screen.

VDU 24

VDU 24,l;b;r;t; will set the printer driver's graphics clipping box. The rectangle specified should lie completely within the box that was reported on return from the last call to `PDriver_DrawPage` or `PDriver_GetRectangle`. If this is not the case, it is not defined what will happen, and different printer

drivers may treat it in different ways. This is analogous to the situation with the window manager. Attempts to set a graphics clipping box outside the rectangle currently being redrawn may be ignored completely if they go outside the screen, or may get obeyed with consequences that are almost certainly wrong.

VDU 29

VDU 29,x;y; sets the printer driver's graphics origin.

VDU 26

VDU 26 will reset the printer driver's graphics clipping box to its maximum size. This is essentially the box reported on return from the last call to PDriver_DrawPage or PDriver_GetRectangle, but may be slightly different due to rounding problems when converting from a box expressed in printer pixels to one expressed in OS units. It also resets its versions of the graphics origin, the current graphics position and all the previous graphics positions to (0,0).

VDU 23,6

VDU 23,6 will fault because dot-dash lines are not implemented in current printer drivers. Use the dashed line facility of Draw_Stroke instead.

VDU 23,16

VDU 23,16,x,y| changes the printer driver's version of the cursor control flags, and thus how the cursor movement control sequences and BBC-style character plotting affect the current graphics position. As usual, this is completely independent of the corresponding flags in the VDU drivers. However, printer drivers pay no attention to the setting of bit 6, which controls whether movements beyond the edge of the graphics window cause carriage return/line feeds and other cursor movements to be generated automatically. They always behave as though it is set. Note that the Wimp normally sets this bit, and that it is not sensible to have it clear at any time during a Wimp redraw.

VDU 23,17,7

VDU 23,17,7,flags,x;y;| changes the printer driver's version of the size that BBC-style characters are to be plotted and the spacing that is required between them. Setting the VDU 4 character size cannot possibly affect the printer driver's output and so will be ignored completely. As noted below under 'Plotting operations', a pixel is regarded as the size of a screen pixel for the screen mode that was in effect when the print job was started.

Plotting operations

The printer driver regards a pixel as having size 2 OS units square (1/90 inch square). The main effect of this is that all PLOT line, PLOT point and PLOT outline calls will produce lines that are approximately 2 OS units wide.

Use Draw module calls if you wish to produce different lines.

VDU 23,17,7

However, when translating the character size and spacing information provided by VDU 23,17,7,... (see above) from pixels to OS units, the screen pixel size for the screen mode that was in effect when the print job was started is used. This is done in the expectation that the application is basing its requested sizes on that screen mode.

VDU plot operations

The following VDU sequences perform straightforward plotting operations; printer drivers will produce the corresponding printed output:

VDU 12	clear graphics window in VDU 5 state
VDU 16	clear graphics window
VDU 25,0-63,x;y;	draw line; however, the lines are always plotted solid, so only VDU 25,0-15,... and VDU 25,32-47,... will look the same as in VDU output. Use Draw_Stroke to generate dashed lines that will come out well in printed output.
VDU 25,64-71,x;y;	draw point
VDU 25,80-87,x;y;	fill triangle
VDU 25,96-103,x;y;	fill axis-aligned rectangle
VDU 25,112-119,x;y;	fill parallelogram
VDU 25,144-151,x;y;	draw circle
VDU 25,152-159,x;y;	fill circle
VDU 25,160-167,x;y;	draw circular arc
VDU 25,168-175,x;y;	fill circular segment
VDU 25,176-183,x;y;	fill circular sector
VDU 25,192-199,x;y;	draw ellipse
VDU 25,200-207,x;y;	fill ellipse
VDU 32-126	print characters in BBC-style font
VDU 127	backspace & delete
VDU 128-255	print characters in BBC-style font

Rounding

One difference to note is that most printer drivers will either not do the rounding to pixel centres normally done by the VDU drivers, or will round to different pixel centres – probably the centres of their device pixels.

Faulted

The following VDU sequences are faulted because they cannot be split up easily across rectangles, and also because they depend on the current picture contents and so cannot be implemented, for example, on PostScript printers:

VDU 25,72-79,x;y;	horizontal line fill (flood fill primitive)
VDU 25,88-95,x;y;	horizontal line fill (flood fill primitive)
VDU 25,104-111,x;y;	horizontal line fill (flood fill primitive)
VDU 25,120-127,x;y;	horizontal line fill (flood fill primitive)
VDU 25,128-143,x;y;	flood fills
VDU 25,184-191,x;y;	copy/move rectangle

VDU 25,184,x;y; and VDU 25,188,x;y; are exceptions to this; they are correctly interpreted by printer drivers as being equivalent to VDU 25,0,x;y; and VDU 25,4,x;y; respectively.

Sprite VDUs

The sprite plotting VDU sequences (VDU 23,27,m,n,l and VDU 25,232-239,x;y;) and the font manager VDU sequences (VDU 23,25,a,b,c,d,e,f,g,h, VDU 23,26,a,b,c,d,e,f,g,h,text and VDU 25,208-215,x;y;text) cannot be handled by the printer drivers and generate errors. You should use OS_SpriteOp and the font manager SWIs instead.

SpriteV

Printer drivers intercept OS_SpriteOp via the SpriteV vector. Most calls are simply passed through to the operating system or the SpriteExtend module. The ones that normally plot to the screen are generally intercepted and used to generate printer output by the printer driver.

Faulted

The following reason codes normally involve reading or writing the screen contents and are not straightforward sprite plotting operations. Because some printer drivers redirect output to a sprite internally, it is unknown what the 'screen' is during these operations. They are therefore faulted.

2	screen save
3	screen load
14	get sprite from current point on screen
16	get sprite from specified point on screen

Passed on

Reason codes that are passed through to the operating system or the SpriteExtend module are:

8	read sprite area control block
9	initialise sprite area
10	load sprite file
11	merge sprite file
12	save sprite file
13	return name of numbered sprite
15	create sprite
25	delete sprite
26	rename sprite
27	copy sprite
29	create mask
30	remove mask
31	insert row
32	delete row
33	flip about X axis
35	append sprite
36	set pointer shape
40	read sprite size
41	read pixel colour
42	write pixel colour
43	read pixel mask
44	write pixel mask
45	insert column
46	delete column
47	flip about Y axis
62	read save area size

Select error

The following reason code is passed through to the operating system when it is called for a user sprite (ie with &100 or &200 added to it), as this call is simply asking the operating system for the address of the sprite concerned. If the system version is called (ie without anything added to it), it is asking for a sprite to be selected for use with the VDU sprite plotting sequences. As these sequences are not handled by the printer driver, this version of the call generates an error.

24	select sprite
----	---------------

Sprite plotting

The following reason codes plot a sprite or its mask, and are converted into appropriate printer output:

28	plot sprite at current point on screen
34	plot sprite at specified point on screen
48	plot mask at current point on screen
49	plot mask at specified point on screen
50	plot mask at specified point on screen, scaled
52	plot sprite at specified point on screen, scaled
53	plot sprite at specified point on screen, grey scaled

Scaled characters

The following reason code is mainly used by the VDU drivers to implement sizes other than 8x8 and 8x16 for VDU 5 characters. It is not handled by the printer drivers, which deal with scaled VDU 5 text by another mechanism, and causes an error if encountered during a print job.

51	plot character, scaled
----	------------------------

GCOLs

As usual for a printer driver, only some GCOL actions are understood. If the GCOL action is not divisible by 8, nothing is plotted. If it is divisible by 8, the overwrite action is used. If it is divisible by 16, the sprite is plotted without using its mask; otherwise the mask is used.

The colours used to plot sprite pixels are determined as follows:

- If the call does not allow a pixel translation table, or if no translation table is supplied, the current screen palette is consulted to find out what RGB combination the sprite pixel's value corresponds to. The printer driver then does its best to produce that RGB combination. Use of this option is not recommended.
- If a translation table is supplied with the call, the printer driver assumes that the table contains code values allocated by one of the following SWIs:

```
ColourTrans_SelectTable with R2 = -1  
ColourTrans_ReturnColourNumber  
ColourTrans_ReturnColourNumberForMode with R1 = -1  
ColourTrans_ReturnOppColourNumber  
ColourTrans_ReturnOppColourNumberForMode with R1 = -1
```

It can therefore look up precisely which RGB combination is supposed to correspond to each sprite pixel value. Because of the variety of ways in which printer drivers can allocate these values, the translation table should always have been set up in the current print job and using these calls.

Scale

If a sprite is printed unscaled, its size on the printed output is the same as its size would be if it were plotted to the screen in the screen mode that was in effect at the time that the print job concerned was started. If it is printed scaled, the scaling factors are applied to this size. This is one of the few ways in which the printed output does depend on this screen mode. The main other ones are in interpreting GCOL and Tint values, and in interpreting VDU 5 character sizes. It is done this way in the expectation that the application is scaling the sprite for what it believes is the current screen mode.

VDU output

Finally, the following two reason codes are intercepted to keep track of whether plotting output is currently supposed to go to a sprite or to the screen. If it is supposed to go to a sprite, it really will go to that sprite.

60	switch output to sprite
61	switch output to mask

This allows applications to create sprites normally while printing. When output is supposed to go to the screen, it will be processed by the printer driver. Note that printer drivers that redirect output to a sprite internally will treat this case specially, regarding output to that sprite as still being destined for the screen.

DrawV

Printer drivers intercept the DrawV vector and re-interpret those calls whose purpose is to plot something on the screen, producing appropriate printer output instead. There are a number of restrictions on the calls that can be dealt with, mainly due to the limitations of PostScript. Most of the operations that are disallowed are not particularly useful, fortunately.

Colour

Note that the Draw module calls normally use the graphics foreground colour to plot with and the graphics origin. The printer driver uses its versions of these values. In particular, this means that the fill colour is subject to all the restrictions noted elsewhere in this document.

Floating point

The floating point Draw module calls are not intercepted at present. If and when the Draw module is upgraded to deal with them, printer drivers will be similarly upgraded.

Draw_Fill

Printer drivers can deal with most common calls to this SWI. The restrictions are:

- They cannot deal with fill styles that invoke the positive or negative winding number rules – ie those with bit 0 set.
- They cannot deal with a fill style which asks for non-boundary exterior pixels to be plotted (ie bit 2 is set), except for the trivial case in which all of bits 2 - 5 are set (ie all pixels in the plane are to be plotted).
- They cannot deal with the following values for bits 5 - 2:
 - 0010 – plot exterior boundary pixels only.
 - 0100 – plot interior boundary pixels only.
 - 1010 – plot exterior boundary and interior non-boundary pixels only.
- An application should not rely on there being any difference between what is printed for the following three values of bits 5 - 2:
 - 1000 – plot interior non-boundary pixels only.
 - 1100 – plot all interior pixels.
 - 1110 – plot all interior pixels and exterior boundary pixels.

A printer driver will generally try its best to distinguish these, but it may not be possible.

Draw_Stroke

Again, most common calls to this SWI can be dealt with. The restrictions on the parameters depend on whether the specified thickness is zero or not.

If the specified thickness is zero, the restrictions are:

- Printer drivers cannot deal with a fill style with bits 3 - 2 equal to 01 – one that asks for pixels lying off the stroke to be plotted and those that lie on the stroke not to be.
- Most printer drivers will not pay any attention to bit 31 of the fill style, which distinguishes plotting the stroke subpath by subpath from plotting it all at once.

If the specified thickness is non-zero, the restrictions are:

- All the restrictions mentioned under Draw_Fill above.
- They cannot deal with bits 5 - 2 being 0110 – a call asking for just the boundary pixels of the resulting filled path to be plotted.
- Most printer drivers will not pay any attention to bit 31 of the fill style, which distinguishes plotting the stroke subpath by subpath from plotting it all at once.

Draw_StrokePath,
Draw_FlattenPath,
Draw_TransformPath

None of these do any plotting; they are all dealt with in the normal way by the Draw module.

Draw_ProcessPath

This SWI is faulted if R7=1 (fill path normally) or R7=2 (fill path subpath by subpath) on entry. Use the appropriate one of Draw_Fill or Draw_Stroke if you want to produce printed output. If the operation you're trying to do is too complicated for them, it almost certainly cannot be handled by the PostScript printer driver for example.

If you are using this call to calculate bounding boxes, using the R7=&80000000 +address option, then the matrix, flatness, line thickness, etc., must exactly correspond with those in the intended call. Because of rounding errors, flattening errors, etc., clipping may result if these parameters are different.

All other values of R7 correspond to calls that don't do any plotting and are dealt with in the normal way by the Draw module. If you're trying to do something complicated and you've got enough workspace and RMA, a possible useful trick is to use Draw_ProcessPath with R7 pointing to an output buffer, followed by Draw_Fill on the result.

ColourV

The printer driver intercepts calls to the ColourTrans module, via the ColourV vector. Most of them are passed straight on to the ColourTrans module. The exceptions are:

ColourTrans_SelectTable with R2 = -1

Each RGB combination in the source palette, or implied by it in the case of 256 colour modes, is converted into a colour number as though by ColourTrans_ReturnColourNumber. The resulting values are placed in the table.

ColourTrans_ SetGCOL

The printer driver's version of the foreground or background colour is set as appropriate. The GCOL actions are interpreted precisely as for the VDU 18,k,c call. However, rather than looking up a GCOL in the screen palette at plot time, the exact RGB combination specified in this call is remembered and used, as accurately as the printer will render it at plot time.

After this has been done, the call is effectively converted into ColourTrans_ReturnGCOL and passed down to the ColourTrans module in order to set the information returned correctly. Note that this implies that subsequently using the GCOL returned in a VDU 18,k,c sequence will not produce the same effect on the colour as this call. It will merely produce the best approximation the printer can manage to the best approximation the current screen palette can manage to the specified RGB combination. It is therefore probably a bad idea to use the values returned.

This call allows the application to make full use of a printer's colour resolution without having to switch to another screen mode or mess around with the screen's palette, and without worrying about the effects of a change in the screen's palette. It is therefore the recommended way to set the foreground and background colours.

ColourTrans_ ReturnColourNumber

This will return a code value, in the range 0-255, that identifies the specified RGB combination as accurately as possible to the printer driver. How this code value is determined may vary from printer driver to printer driver, and indeed even from print job to print job for the same printer driver. An application should therefore not make any assumptions about what these code values mean. Most printer drivers implement this by pre-allocating some range of code values to evenly spaced RGB combinations, then adopting the following approach:

- If the RGB combination is already known about, return the corresponding code value.
- If the RGB combination is not already known about and some code values are still free, allocate one of the unused code values to the new RGB combination and return that code value.
- If the RGB combination is not already known about and all code values have been allocated, return the code number whose RGB combination is as close as possible to the desired RGB combination.

The pre-allocation of evenly spaced RGB combinations will ensure that even the third case does not have really terrible results.

ColourTrans_
ReturnColourNumberFor
Mode with R1 = -1

This is treated exactly the same as ColourTrans_ReturnColourNumber above.

'Opposite' colours

The printer driver handles 'opposite' colours in a subtly different way to the ColourTrans module. It returns the colour closest to the RGB value most different to that given, whereas ColourTrans returns the colour furthest from the given RGB. This difference will only be obvious if your printer cannot print a very wide range of colours.

ColourTrans_
SetOppGCOL

This behaves like ColourTrans_SetGCOL above, except that the RGB combination it remembers is the furthest possible RGB combination from the one actually specified in R0, and it ends by being converted into a call to ColourTrans_ReturnOppGCOL. Note that there is no guarantee that the GCOL returned is anywhere near the RGB combination remembered.

ColourTrans_ReturnOpp
ColourNumber

This behaves exactly as though ColourTrans_ReturnColourNumber had been called with R0 containing the furthest possible RGB combination from the one actually specified.

ColourTrans_ReturnOpp
ColourNumber
ForMode with R1 = -1

This behaves exactly as though ColourTrans_ReturnColourNumberForMode (see above) had been called with R1 = -1 and R0 containing the furthest possible RGB combination from the one actually specified.

ColourTrans_
SetFontColours

The printer driver's version of the font colours is set, to as accurate a representation of the desired RGB combinations as the printer can manage.

Before this is done, the call is passed down to the ColourTrans module to determine the information to be returned. Note that this implies that subsequently using the values returned in a Font_SetFontColours call will not produce the same effect on the font colours as this call. It will merely produce the best approximations the printer can manage to the best approximations the current screen palette can manage to the specified RGB combinations. It is therefore a bad idea to use the values returned.

This call therefore allows the application to make full use of a printer's colour resolution without having to switch to another screen mode or mess around with the screen's palette, and without worrying about the effects of a change in the screen's palette. It is the recommended way to set the font colours.

Font manager SWIs

The printer driver interacts with the font manager via a service call and PDriver_FontSWI in such a way that when it is active, calls to the following SWIs are processed by the printer driver:

- Font_Paint
- Font_LoseFont
- Font_SetFontColours
- Font_SetPalette

This enables the printer driver to make Font_Paint produce printer output rather than affecting the screen.

Font_SetFontColours

The use of Font_SetFontColours is not recommended, as it results in the setting of colours that depend on the current screen palette. Instead, use ColourTrans_SetFontColours to set font colours to absolute RGB values. Similarly, the use of colour-changing control sequences in strings passed to Font_Paint is not recommended.

Font_Paint

How exactly this call operates varies quite markedly between printer drivers. For instance, most dot matrix printer drivers will probably use the font manager to write into the sprite they are using to hold the current strip of printed output, while the PostScript printer driver uses the PostScript prologue to define a translation from font manager font names to printer fonts.

Miscellaneous SWIs

OS_Byte 163

OS_Byte 163,242,0-64 are intercepted to set the printer driver version of the dot pattern repeat length instead of the VDU drivers' version.

OS_Byte 218

OS_Byte 218 is intercepted to act on the printer driver's VDU queue instead of the VDU drivers' version.

OS_ReadVduVariables

It should be noted that most of the informational calls associated with the VDU drivers, and OS_ReadVduVariables in particular, will produce undefined results when a printer driver is active. These results are likely to differ between printer drivers. In particular, they will vary according to whether the printer driver plots to a sprite internally and if so, how large the sprite concerned is.

The only informational calls that the application may rely upon are:

OS_Word 10	used to read character and ECF definitions.
OS_Word 11	used to read palette definitions.
OS_ReadPalette	used to read palette definitions.
OS_Byte 218	when used to read number of bytes in VDU queue.

Error handling

There are a couple of somewhat unusual features about the printer drivers' error handling that an application author should be aware of:

Escape handling

First, Escape condition generation and side effects are turned on within various calls to the printer driver and restored to their original state afterwards. If the application has Escape generation turned off, it is guaranteed that any Escape generated within the print job will be acknowledged and turned into an 'Escape' error. If the application has Escape generation turned on, most Escapes generated within the print job will be acknowledged and turned into 'Escape' errors, but there is a small period at the end of the call during which an Escape will not be acknowledged. If the application makes a subsequent call of one of the relevant types to the printer driver, that subsequent call will catch the Escape. If no such subsequent call is made, the application will need to trap the Escape itself.

The SWIs during which Escape generation is turned on are:

- PDriver_SelectJob for a new job
- PDriver_EndJob
- OS_WriteC
- All ColourTrans SWIs – except ColourTrans_GCOLTocolourNumber, ColourTrans_ColourNumberToGCAL, and ColourTrans_InvalidCache.
- Draw_Fill
- Draw_Stroke
- Font_SetFontColours
- Font_SetPalette
- Font_Paint
- OS_SpriteOp with reason codes:
 - PutSprite
 - PutSpriteUserCoords
 - PutSpriteScaled
 - PutSpriteGreyScaled
 - PlotMask
 - PlotMaskUserCoords
 - PlotMaskScaled

All but the first two only apply at times when the printer driver is intercepting plotting calls; that is, at times when all of the following conditions hold:

- There is an active print job.
- Plotting output is directed either to the screen or to a sprite internal to the printer driver.
- The Wimp is not reporting an error. This is as defined by the service call with reason WimpReportError.

Persistent errors

Secondly, inside a number of calls, any error that occurs is converted into a "persistent error". The net effect of this is that:

- The error number is left unchanged.
- The error message has the string " (print cancelled)" appended to it. If it is so long that this would cause it to exceed 255 characters, it is truncated to a suitable length and "... (print cancelled)" is appended to it.
- Any subsequent call to any of the routines concerned will immediately return the same error.

The reason for this behaviour is to prevent errors the application is not expecting from being ignored; for example, quite a lot of code assumes incorrectly that OS_WriteC cannot produce an error. This ensures that an error during OS_WriteC cannot reasonably get ignored forever.

The SWIs during which persistent errors are created are:

- PDriver_EndJob
- PDriver_GiveRectangle
- PDriver_DrawPage
- PDriver_GetRectangle
- OS_WriteC
- All ColourTrans SWIs – except ColourTrans_GCOLToColourNumber, ColourTrans_ColourNumberToGCAL, and ColourTrans_InvalidCache.
- Draw_Fill
- Draw_Stroke
- Draw_ProcessPath with R7=1
- Font_SetFontColours
- Font_SetPalette
- Font_Paint

- OS_SpriteOp with reason codes:
 - PutSprite
 - PutSpriteUserCoords
 - PutSpriteScaled
 - PutSpriteGreyScaled
 - PlotMask
 - PlotMaskUserCoords
 - PlotMaskScaled
 - ScreenSave
 - ScreenLoad
 - GetSprite
 - GetSpriteUserCoords
 - PaintCharScaled
 - SelectSprite in the system sprite area only
 - Reason codes unknown to the printer driver

All but the first four only apply at times that the printer driver is intercepting plotting calls. See above for details of this.

PDriver_CancelJob

PDriver_CancelJob puts a print job into a similar state, with the error message being simply "Print cancelled". However, this error is only returned by subsequent calls from the list above, not by PDriver_CancelJob itself.

PDriver_AbortJob

Note that an application must respond to any error during a print job that could have come from one of the above sources by calling PDriver_AbortJob. In particular, take care to respond to errors from PDriver_EndJob by calling PDriver_AbortJob, not PDriver_EndJob, otherwise an infinite succession of errors will occur or an unfinished print job will be left around.

SWI Calls

PDriver_Info (SWI &80140)

Get information on the printer driver

On entry

—

On exit

R0 = general type of printer chosen and version number of driver (see below)

R1 = x resolution of printer driven in dots per inch

R2 = y resolution of printer driven in dots per inch

R3 = features word (see below)

R4 = pointer to string containing adjectival description of printers supported

R5 = x halftone resolution in repeats/inch. Same as R1 if no halftoning

R6 = y halftone resolution in repeats/inch. Same as R2 if no halftoning

R7 = printer driver specific number identifying the configured printer

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI tells an application what the capabilities of the attached printer are. This allows the application to change the way it outputs its data to suit the printer.

Some of the values returned can be changed by the configuration application attached to the printer driver by PDriver_SetInfo.

If this is called while a print job is selected, the values returned are those that were in effect when that print job was started with PDriver_SelectJob. If it is called when no print job is active, the values returned are those that would be used for a new print job.

The value returned in R0 is split in half. The top 16 bits contains the description of which printer driver type is running. The current values it can have are:

- 0 = PostScript
- 1 = FX80 or similar
- 2 = HP LaserJet or compatible
- 3 = Integrex ColourJet

The bottom 16 bits of R0 have the version number of the printer driver times 100. eg. Version 3.21 would be 321 (&0141).

R3 returns a bitfield that describes the available features of the current printer. Most applications shouldn't need to look at this word, unless they wish to alter their output depending on the facilities available.

It is split into several fields:

Bits	Subject
0 - 7	printer driver's colour capabilities
8 - 15	printer driver's plotting capabilities
16 - 23	reserved – must be set to zero
24 - 31	printer driver's optional features

In more detail, each individual bit has the following meaning:

Bit(s)	Value	Meaning
0	0	it can only print in monochrome.
	1	it can print in colour.
1	0	it supports the full colour range – ie it can manage each of the eight primary colours. Ignored if bit 0 = 0.
	1	it supports only a limited set of colours.
2	0	it supports a semi-continuous range of colours at the software level. Also, if bit 0 = 0 and bit 2 = 0, then an application can expect to plot in any level of grey.
	1	it only supports a discrete set of colours at the software level; it does not support mixing, dithering, toning or any similar technique.
3 - 7		reserved and set to zero.

8	0	it can handle filled shapes.
	1	it cannot handle filled shapes other than by outlining them; an unsophisticated XY plotter would have this bit set, for example.
9	0	it can handle thick lines.
	1	it cannot handle thick lines other than by plotting a thin line. An unsophisticated XY plotter would also come into this category. The difference is that the problem can be solved, at least partially, if the plotter has a range of pens of differing thicknesses available.
10	0	it handles overwriting of one colour by another on the paper properly. This is generally true of any printer that buffers its output, either in the printer or the driver.
	1	it does not handle overwriting of one colour by another properly, but only overwriting of the background colour by another. This is a standard property of XY plotters.
11 - 15		reserved and set to zero.
16 - 23		reserved and set to zero.
24	0	it does not support screen dumps.
	1	it does support screen dumps.
25	0	it does not support transformations other than scalings, translations, rotations by multiples of 90 degrees and combinations thereof. These are the transformations supplied to PDriver_DrawPage.
	1	it does support arbitrary transformations supplied to PDriver_DrawPage.
26	0	it does not support the PDriver_InsertIllustration call
	1	it does support the PDriver_InsertIllustration call
27 - 31		reserved and set to zero.

The table below shows the effect of bits 0 - 2 in more detail:

Bit 0	Bit 1	Bit 2	Colours available
0	0	0	Arbitrary greys
0	0	1	A limited set of greys (probably only black and white)
0	1	0	Arbitrary greys
0	1	1	A limited set of greys (probably only black and white)
1	0	0	Arbitrary colours
1	0	1	A limited set of colours, including all the eight primary colours
1	1	0	Arbitrary colours within a limited range (for example, it might be able to represent arbitrary greys, red, pinks and so on, but no blues or greens). This is not a very likely option
1	1	1	A finite set of colours - as for instance an XY plotter might have

Related SWIs

None

Related vectors

None

PDriver_SetInfo (SWI &80141)

Configure the printer driver

On entry

R1 = x resolution of printer driven in dots per inch
R2 = y resolution of printer driven in dots per inch
R3 = bit 0 only is used— all other bits are ignored
R5 = x halftone resolution in repeats/inch (same as R1 if no halftoning)
R6 = y halftone resolution in repeats/inch (same as R2 if no halftoning)
R7 = printer driver specific number identifying the configured printer

On exit

R1 - R3 preserved
R5 - R7 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call is used by the printer driver configuration application on the desktop to set the user requested settings.

It must never be called by any other application.

Related SWIs

PDriver_Info (SWI &80140)

Related vectors

None

PDriver_CheckFeatures (SWI &80142)

Check the features of a printer

On entry

R0 = features word mask
R1 = features word value

On exit

R0, R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

If the features word that PDriver_Info would return in R3 satisfies $((\text{features word}) \text{ AND } R0) = (R1 \text{ AND } R0)$, then the return is normal with all registers preserved. Otherwise a suitable error is generated if appropriate. For example, no error will be generated if the printer driver has the ability to support arbitrary rotations and your features word value merely requests axis preserving ones.

Related SWIs

PDriver_Info (SWI &80140)

Related vectors

None

PDriver_PageSize (SWI &80143)

Find how large paper and print area is

On entry

—

On exit

R1 = x size of paper, including margins
R2 = y size of paper, including margins
R3 = left edge of printable area of paper
R4 = bottom edge of printable area of paper
R5 = right edge of printable area of paper
R6 = top edge of printable area of paper

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

An application can use this call to find out how big the paper in use is and how large the printable area on the paper is. This information can then be used to decide how to place the data to be printed on the page.

These values can be changed by the configuration application associated with the printer driver (using PDriver_SetPageSize). If PDriver_PageSize is called while a print job is selected, the values returned are those that were in effect when that print job was started (ie. when it was first selected using PDriver_SelectJob). If PDriver_PageSize is called when no print job is active, the values returned are those that would be used for a new print job.

All units are in millipoints, and R3 - R6 are relative to the bottom left corner of the page.

Related SWIs

PDriver_SetPageSize (SWI &80144)

Related vectors

None

PDriver_SetPageSize (SWI &80144)

Set how large paper and print area is

On entry

R1 = x size of paper, including margins
R2 = y size of paper, including margins
R3 = left edge of printable area of paper
R4 = bottom edge of printable area of paper
R5 = right edge of printable area of paper
R6 = top edge of printable area of paper

On exit

R1 - R6 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The configuration application associated with a particular printer driver uses this SWI to change the page size values associated with subsequent print jobs.

It must never be called by any other application.

All units are in millipoints, and R3 - R6 are relative to the bottom left corner of the page.

Related SWIs

PDriver_PageSize (SWI &80143)

Related vectors

None

PDriver_SelectJob (SWI &80145)

Make a given print job the current one

On entry

R0 = file handle for print job to be selected, or zero to cease having any print job selected.

R1 = zero or points to a title string for the job

On exit

R0 = file handle for print job that was previously active, or zero if no print job was active.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

A print job is identified by a file handle, which must be that of a file that is open for output. The printer output for the job concerned is sent to this file.

Calling PDriver_SelectJob with R0=0 will cause the current print job (if any) to be suspended, and the printer driver will cease intercepting plotting calls.

Calling PDriver_SelectJob with R0 containing a file handle will cause the current print job (if any) to be suspended, and a print job with the given file handle to be selected. If a print job with this file handle already exists, it is resumed; otherwise a new print job is started. The printer driver will start to intercept plotting calls if it is not already doing so.

Note that this call never ends a print job. To do so, use one of the SWIs PDriver_EndJob or PDriver_AbortJob.

The title string pointed to by R1 is treated by different printer drivers in different ways. It is terminated by any character outside the range ASCII 32 - 126. It is only ever used if a new print job is being started, not when an old one is being resumed. Typical uses are:

- A simple printer driver might ignore it.

- The PostScript printer driver adds a line "%Title: " followed by the given title string to the PostScript header it generates.
- Printer drivers whose output is destined for an expensive central printer in a large organisation might use it when generating a cover sheet for the document.

An application is always entitled not to supply a title (by setting R1=0), and a printer driver is entitled to ignore any title supplied.

Printer drivers may also use the following OS variables when creating cover sheets, etc:

PDriver\$For	indicates who the output is intended to go to
PDriver\$Address	indicates where to send the output.

These variables must not contain characters outside the range ASCII 32 - 126.

If an error occurs during PDriver_SelectJob, the previous job will still be selected afterwards, though it may have been de-selected and re-selected during the call. No new job will exist. One may have been created during the call, but the error will cause it to be destroyed again.

Related SWIs

PDriver_CurrentJob (SWI &80146), PDriver_EndJob (SWI &80148),
PDriver_AbortJob (SWI &80149), PDriver_Reset (SWI &8014A)

Related vectors

None

PDriver_CurrentJob (SWI &80146)

	Get the file handle of the current job
On entry	—
On exit	R0 = file handle
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	R0 returns the file handle for the current active print job, or zero if no print job is active.
Related SWIs	PDriver_SelectJob (SWI &80145), PDriver_EndJob (SWI &80148), PDriver_AbortJob (SWI &80149), PDriver_Reset (SWI &8014A)
Related vectors	None

PDriver_FontSWI (SWI &80147)

	Internal call
On entry	—
On exit	—
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This SWI is part of the internal interface between the font system and printer drivers. Applications must not call it.
Related SWIs	None
Related vectors	None

PDriver_EndJob (SWI &80148)

	End a print job normally
On entry	R0 = file handle for print job to be ended
On exit	R0 = preserved
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	<p>This SWI should be used to end a print job normally. This may result in further printer output – for example, the PostScript printer driver will produce the standard trailer comments.</p> <p>If the print job being ended is the currently active one, there will be no current print job after this call, so plotting calls will no longer be intercepted.</p> <p>If the print job being ended is not currently active, it will be ended without altering which print job is currently active or whether plotting calls are being intercepted.</p>
Related SWIs	PDriver_SelectJob (SWI &80145), PDriver_CurrentJob (SWI &80146), PDriver_AbortJob (SWI &80149), PDriver_Reset (SWI &8014A)
Related vectors	None

PDriver_AbortJob (SWI &80149)

	End a print job without any further output
On entry	R0 = file handle for print job to be aborted
On exit	R0 = preserved
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	<p>This SWI should be used to end a print job abnormally. It should generally be called after errors while printing. It will not try to produce any further printer output. This is important if an error occurs while sending output to the print job's output file.</p> <p>If the print job being aborted is the currently active one, there will be no current print job after this call, so plotting calls will no longer be intercepted.</p> <p>If the print job being aborted is not currently active, it will be aborted without altering which print job is currently active or whether plotting calls are being intercepted.</p>
Related SWIs	PDriver_SelectJob (SWI &80145), PDriver_CurrentJob (SWI &80146), PDriver_EndJob (SWI &80148), PDriver_Reset (SWI &8014A)
Related vectors	None

PDriver_Reset (SWI &8014A)

	Abort all print jobs
On entry	—
On exit	—
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	<p>This SWI aborts all print jobs known to the printer driver, leaving the printer driver with no active or suspended print jobs and ensuring that plotting calls are not being intercepted.</p> <p>Normal applications shouldn't use this SWI, but it can be useful as an emergency recovery measure when developing an application.</p> <p>A call to this SWI is generated automatically if the machine is reset or the printer driver module is killed or RMTidyed.</p>
Related SWIs	PDriver_SelectJob (SWI &80145), PDriver_CurrentJob (SWI &80146), PDriver_EndJob (SWI &80148), PDriver_Abort (SWI &80149)
Related vectors	None

PDriver_GiveRectangle (SWI &8014B)

Specify a rectangle to be printed

On entry

R0 = rectangle identification word
R1 = pointer to 4 word block, containing rectangle to be plotted in OS units.
R2 = pointer to 4 word block, containing transformation table
R3 = pointer to 2 word block, containing the plot position.
R4 = background colour for this rectangle, in the form &BBGRRXX.

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI allows an application to specify a rectangle from its workspace to be printed, how it is to be transformed and where it is to appear on the printed page.

The word in R0 is reported back to the application when it is requested to plot all or part of this rectangle.

The value passed in R2 is the dimensionless transformation to be applied to the specified rectangle before printing it. The entries are given as fixed point numbers with 16 binary places, so the transformation is:

$$x' = (x * R2!0 + y * R2!8) / 2^{16}$$

$$y' = (x * R2!4 + y * R2!12) / 2^{16}$$

The value passed in R3 is the position where the bottom left corner of the rectangle is to be plotted on the printed page in millipoints.

An application should make one or more calls to PDriver_GiveRectangle before a call to PDriver_DrawPage and the subsequent calls to PDriver_GetRectangle. Multiple calls allow the application to print multiple rectangles from its workspace to one printed page – for example, for "two up" printing.

The printer driver may subsequently ask the application to plot the specified rectangles or parts thereof in any order it chooses. An application should not make any assumptions about this order or whether the rectangles it specifies will be split. A common reason why a printer driver might split a rectangle is that it prints the page in strips to avoid using excessively large page buffers.

Assuming that a non-zero number of copies is requested and that none of the requested rectangles go outside the area available for printing, it is certain to ask the application to plot everything requested at least once. It may ask for some areas to be plotted more than once, even if only one copy is being printed, and it may ask for areas marginally outside the requested rectangles to be plotted. This can typically happen if the boundaries of the requested rectangles are not on exact device pixel boundaries.

If PDriver_GiveRectangle is used to specify a set of rectangles that overlap on the output page, the rectangles will be printed in the order of the PDriver_GiveRectangle calls. For appropriate printers (ie. most printers, but not XY plotters for example), this means that rectangles supplied via later PDriver_GiveRectangle calls will overwrite rectangles supplied via earlier calls.

The rectangle specified should be a few OS units larger than the 'real' rectangle, especially if important things lie close to its edge. This is because rounding errors are liable to appear when calculating bounding boxes, resulting in clipping of the image. Such errors tend to be very noticeable, even when the amounts involved are small.

However, you shouldn't make the rectangle a lot larger than the real rectangle. This will result in slowing the process down and use of unnecessarily large amounts of memory. Also, some subsequent users may scale the image according to this rectangle size (say to use some PostScript as an illustration in another document), resulting in it being too small.

Related SWIs

PDriver_GetRectangle (SWI &8014D)

Related vectors

None

PDriver_DrawPage (SWI &8014C)

Called after all rectangles plotted to draw the page

On entry

R0 = number of copies to print
R1 = pointer to 4 word block, to receive the rectangle to print
R2 = page sequence number within the document, or 0
R3 = zero or points to a page number string

On exit

R0 = non-zero if rectangle required, zero if finished
R1 = preserved
R2 = rectangle identification word if R0 is non-zero
R3 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI should be called after all rectangles to be plotted on the current page have been specified using PDriver_GiveRectangle. It returns the first rectangle that the printer driver wants plotted in the area. If nothing requires plotting it will indicate the end of the list.

R2 on entry is zero or contains the page's sequence number within the document being printed (ie. 1-n for an n-page document).

R3 on entry is zero or points to a string, terminated by a character in the ASCII range 33 - 126, which gives the text page number: for example "23", "viii", "A-1". Note that spaces are not allowed in this string.

If R0 on exit is non-zero, the area pointed to by R1 has been filled in with the rectangle that needs to be plotted, and R2 contains the rectangle identification word for the user-specified rectangle that this is a part of. If R0 is zero, the contents of R2 and the area pointed to by R1 are undefined. The rectangle in R1 is in user coordinates before transformation.

The application should stop trying to plot the current page if $R0=0$, and continue otherwise. If $R0 \neq 0$, the fact that $R0$ is the number of copies still to be printed is only intended to be used for information purposes – for example, putting a "Printing page m of n " message on the screen. Note that on some printer drivers, you cannot rely on this number changing incrementally. i.e. it may suddenly go from ' n ' to zero. As long as it is non-zero, $R0$'s value does not affect what the application should try to plot.

The information passed in $R2$ and $R3$ is not particularly important, though it helps to make output produced by the PostScript printer driver conform better to Adobe's structuring conventions. If non-zero values are supplied, they should be correct. Note that $R2$ is NOT the sequence number of the page in the print job, but in the document.

An example: if a document consists of 11 pages, numbered "" (the title page), "i"–"iii" and "1"–"7", and the application is requested to print the entire preface part, it should use $R2 = 2, 3, 4$ and $R3 \rightarrow "i", "ii", "iii"$ for the three pages.

When plotting starts in a rectangle supplied by a printer driver, the printer driver behaves as though the VDU system is in the following state:

- VDU drivers enabled.
- VDU 5 state has been set up.
- all graphics cursor positions and the graphics origin have been set to (0,0) in the user's rectangle coordinate system.
- a VDU 5 character size and spacing of 16 OS units by 32 OS units have been set in the user's rectangle coordinate system.
- the graphics clipping region has been set to bound the actual area that is to be plotted. But note that an application cannot read what this area is: the printer drivers do not and cannot intercept `OS_ReadVduVariables` or `OS_ReadModeVariable`.
- the area in which plotting will actually take place has been cleared to the background colour supplied in the corresponding `PDriver_GiveRectangle` call, as though the background had been cleared.

- the cursor movement control bits (ie the ones that would be set by VDU 23,16,...) are set to &40 – so that cursor movement is normal, except that movements beyond the edge of the graphics window in VDU 5 mode do not generate special actions.
- one OS unit on the paper has a nominal size of 1/180 inch, depending on the transformation supplied with this rectangle.

This is designed to be as similar as possible to the state set up by the window manager when redrawing.

Related SWIs

None

Related vectors

None

PDriver_GetRectangle (SWI &8014D)

Get the next print rectangle

On entry

R1 = pointer to 4 word block, to receive the print rectangle

On exit

R0 = number of copies still requiring printing, or zero if no more plotting

R1 = preserved

R2 = rectangle identification word if R0 is non-zero

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI should be used after plotting a rectangle returned by a previous call to PDriver_DrawPage or PDriver_GetRectangle, to get the next rectangle the printer driver wants plotted. It returns precisely the same information as PDriver_DrawPage.

If R0 is non-zero, the area pointed to by R1 has been filled in with the rectangle that needs to be plotted, and R2 contains the rectangle identification word for the user-specified rectangle that this is a part of. If R0 is zero, the contents of R2 and the area pointed to by R1 are undefined.

Related SWIs

None

Related vectors

None

PDriver_CancelJob (SWI &8014E)

Stops the print job associated with a file handle from printing

On entry

R0 = file handle for job to be cancelled

On exit

R0 = preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI causes subsequent attempts to output to the print job associated with the given file handle to do nothing other than generate the error "Print cancelled". An application is expected to respond to this error by aborting the print job. Generally, this call is used by applications other than the one that started the job.

Related SWIs

PDriver_AbortJob (SWI &80149)

Related vectors

None

PDriver_ScreenDump (SWI &8014F)

	Output a screen dump to the printer
On entry	R0 = file handle of file to receive the dump
On exit	R0 = preserved
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	If this SWI is supported (ie. if bit 24 is set in the value PDriver_Info returns in R3), this SWI causes the printer driver to output a screen dump to the file handle supplied in R0. The file concerned should be open for output. If the SWI is not supported, an error is returned.
Related SWIs	None
Related vectors	None

PDriver_EnumerateJobs (SWI &80150)

List existing print jobs

On entry

R0 = zero to get first, or previous handle to get next print job handle

On exit

R0 = next print job handle, or zero if there are no more in the list

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This allows the print jobs that exist to be enumerated. The order in which they appear is undefined.

Related SWIs

None

Related vectors

None

PDriver_SetPrinter (SWI &80151)

	Set printer driver specific options
On entry	Printer driver specific
On exit	Printer driver specific
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor Mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This allows the setting of options specific to a particular printer driver. In general, this SWI is used by the configuration application associated with the printer driver module and no other application should use it.
Related SWIs	None
Related vectors	None

PDriver_CancelJobWithError (SWI &80152)

Cancels a print job – future attempts to output to it generate an error

On entry

R0 = file handle for job to be cancelled
R1 = pointer to error block

On exit

—

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI causes subsequent attempts to output to the print job associated with the given file handle to do nothing other than generate the specified error. An application is expected to respond to this error by aborting the print job.

This SWI only exists in versions 2.00 and above of the printer driver module (which is present in versions 1.00 and above of the printer driver application).

Related SWIs

None

Related vectors

None

PDriver_SelectIllustration (SWI &80153)

Makes the given print job the current one, and treats it as an illustration

On entry

R0 = file handle for print job to be selected, or 0 to deselect all jobs

R1 = pointer to title string for job, or 0

On exit

R0 = file handle for previously active print job, or 0 if none was active

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call does exactly the same thing as PDriver_SelectJob, except when it used to start a new print job. In this case, the differences are:

- The print job started must contain exactly one page; if it doesn't, an error will be generated.
- Depending on the printer driver involved, the output generated may differ. (For instance, the PostScript printer driver will generate Encapsulated PostScript output for a job started this way.)

The intention of this SWI is that it should be used instead of PDriver_SelectJob when an application is printing a single page that is potentially useful as an illustration in another document.

This SWI only exists in versions 2.00 and above of the printer driver module (which is present in versions 1.00 and above of the printer driver application).

Related SWIs

None

Related vectors

None

PDriver_InsertIllustration (SWI &80154)

Inserts a file containing an illustration into the current job's output

On entry

R0 = file handle for file containing illustration.

R1 = pointer to Draw module path to be used as a clipping path, or 0 if no clipping is required.

R2 = x coordinate of where the bottom left corner of the illustration is to go.

R3 = y coordinate of where the bottom left corner of the illustration is to go.

R4 = x coordinate of where the bottom right corner of the illustration is to go.

R5 = y coordinate of where the bottom right corner of the illustration is to go.

R6 = x coordinate of where the top left corner of the illustration is to go.

R7 = y coordinate of where the top left corner of the illustration is to go.

On exit

—

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

If this SWI is supported (bit 26 is set in the value SWI_PDriver_Info returns in R3), it allows an external file containing an illustration, such as an Encapsulated PostScript file, to be inserted into the current job's output. The format of such an illustration file depends on the printer driver concerned, and many printer drivers won't support any sort of illustration file inclusion at all.

All coordinates in the clipping path and in R2 - R7 are in 256ths of an OS unit, relative to the PDriver_GiveRectangle rectangle currently being processed.

This SWI only exists in versions 2.00 and above of the printer driver module (which is present in versions 1.00 and above of the printer driver application).

Related SWIs

None

Related vectors

None

Application Notes

This is an example BASIC procedure that does a standard "two up" printing job:

```
DEFPROCprintout(firstpage%, lastpage%, title$, filename$)
REM Get SWI numbers used in this procedure.
LOCAL select%, abort%, pagesize%, giverect%, drawpage%, getrect%, end%
SYS "OS_SWINumberFromString",,"PDriver_SelectJob" TO select%
SYS "OS_SWINumberFromString",,"PDriver_AbortJob" TO abort%
SYS "OS_SWINumberFromString",,"PDriver_PageSize" TO pagesize%
SYS "OS_SWINumberFromString",,"PDriver_GiveRectangle" TO giverect%
SYS "OS_SWINumberFromString",,"PDriver_DrawPage" TO drawpage%
SYS "OS_SWINumberFromString",,"PDriver_GetRectangle" TO getrect%
SYS "OS_SWINumberFromString",,"PDriver_EndJob" TO end%
:
REM Open destination file and set up a local error handler that
REM will close it again on an error.
LOCAL H%, O%
H% = OPENOUT(filename$)
LOCAL ERROR
ON ERROR LOCAL:RESTORE ERROR:CLOSE#H%:PROCpasserror
:
REM Start up a print job associated with this file, remembering the
REM handle associated with the previous print job (if any), then
REM set up a local error handler for it.
SYS select%,H%,title$ TO O%
LOCAL ERROR
ON ERROR LOCAL:RESTORE ERROR:SYSabort%,H%:SYSselect%,O%:PROCpasserro
:
REM Now we decide how two pages are to fit on a piece of paper.
LOCAL left%, bottom%, right%, top%
PROCgetdocumentsize(box%)
SYS pagesize% TO ,,left%,bottom%,right%,top%
PROCFittwopages(left%,bottom%,right%,top%,box%,matrix%,origin1%,origin2%)
:
REM Start the double page loop
LOCAL page%, copiesleft%, pagetoprint%, white%
white%=&FFFFFF00
:
FOR page%=firstpage% TO lastpage% STEP 2
:
REM Set up to print two pages, or possibly just one last time around.
SYS giverect%, page%, box%, matrix%, origin1%, white%
IF page%<lastpage% THEN
SYS giverect%, page%+1, box%, matrix%, origin2%, white%
ENDIF
:
REM Start printing. As each printed page corresponds to two document pages,
REM we cannot easily assign any sensible page numbers to printed pages.
REM So we simply pass zeroes to PDriver_DrawPage.
SYS drawpage%,1,box2%,0,0 TO copiesleft%, ,pagetoprint%
WHILE copiesleft%<>0
PROCprintpage(pagetoprint%, box2%)
SYS getrect%, ,box% TO copiesleft%, ,pagetoprint%
```

```

ENDWHILE
:
REM End of page loop
NEXT
:
REM All pages have now been printed. Terminate this print job.
SYS end%,H%
:
REM Go back to the first of our local error handlers.
RESTORE ERROR
:
REM And go back to whatever print job was active on entry to this procedure
REM (or to no print job in no print job was active).
SYS select%,O%
:
REM Go back to the caller's error handler.
RESTORE ERROR
REM Close the destination file.
CLOSE#H%
ENDPROC
:
DEFPROCpasserror
ERROR ERR,REPORT$+" (from line "+STR$(ERL)+")"
ENDPROC

```

This uses the following global areas of memory:

box%	4 words
box2%	4 words
matrix%	4 words
origin1%	2 words
origin2%	2 words

And the following external procedures:

```
DEFPROCgetdocumentsize(box%)
```

- fills the area pointed to by box% with the size of a document page in OS units.

```
DEFPROCfittwopages(l%, b%, r%, t%, box%, transform%, org1%, org2%)
```

- given left, bottom, right and top bounds of a piece of paper, and a bounding box of a document page in OS units, sets up a transformation and two origins in the areas pointed to by tr%, org1% and org2% to print two of those pages on a piece of paper.

```
DEFPROCdrawpage (page%, box%)
```

- draw the parts of document page number 'page%' that lie with the box held in the 4 word area pointed to by 'box%'.

If printing is likely to take a long time and the application does not want to hold other applications up while it prints, it should regularly use a sequence like the following during printing:

```
SYS select%,0%  
SYS "Wimp_Poll",mask%,area% TO reason%  
...  
<process reason% as appropriate>  
...  
SYS select%,H% TO 0%
```


The Sound system

Introduction

The Sound system provides facilities to synthesise and playback high quality digital samples of sound. Since any sound can be stored digitally, the system can equally well generate music, speech and sound effects. Eight fully independent channels are provided.

The sound samples are synthesised in real time by software. A range of different Voice Generators generate a standard set of samples, to which further ones can be added. The software also includes the facility to build sequences of notes.

The special purpose hardware provided on ARM-based systems simply reads samples at a programmable rate and converts them to an analogue signal. Filters and mixing circuitry on the main board provide both a stereo output (suitable for driving personal hi-fi stereo headphones directly, or connecting to an external hi-fi amplifier) and a monophonic or stereophonic output to the internal speaker(s).

Overview

There are four parts to the software for the Sound system: the DMA Handler, the Channel Handler, the Scheduler, and Voice Generators. These are briefly summarised below, and described in depth in later sections.

The DMA Handler

The DMA Handler manages the DMA buffers used to store samples of sound, and the associated hardware used.

The system uses two buffers of digital samples, stored as signed logarithms. The data from one buffer is read and converted to an analogue signal, while data is simultaneously written to the other buffer by a Voice Generator. The two buffers are then swapped between, so that each buffer is successively written to, then read.

The DMA Handler is activated every time a new buffer of sound samples is required. It sends a Fill Request to the Channel Handler, asking that the correct Voice Generators fill the buffer that has just been read from.

The DMA Handler also provides interfaces to program hardware registers used by the Sound system. The number of channels and the stereo position of each one can be set, the built-in loudspeaker(s) can be enabled or disabled, and the entire Sound system can also be enabled or disabled. The sample length and sampling rate can also be set.

The services of the DMA Handler are mainly provided in firmware requiring privileged supervisor status to program the system devices. It is tightly bound to the Channel Handler, sharing static data space. Consequently, this module must not be replaced or amended independently of the Channel Handler.

The Channel Handler

The Channel Handler provides interfaces to control the sound produced by each channel, and maintains internal tables necessary for the rest of the Sound system to produce these sounds.

The interfaces can be used to set the overall volume and tuning, to attach the channels to different Voice Generators, and to start sounds with given pitch, amplitude and duration.

The following internal tables are built and maintained: a mapping of voice names to internal voice numbers; a record for each channel of its volume, voice, pitch and timbre; and linear and logarithmic lookup tables for Voice Generators to scale their amplitude to the current overall volume setting.

Fill Requests issued by the DMA Handler are routed through the Channel Handler to the correct Voice Generators. This allows any tables involved to be updated.

The Channel Handler is tightly bound to the DMA Handler, sharing static data space. Consequently, this module must not be replaced or amended independently of the DMA Handler

The Scheduler

The Scheduler is used to queue Sound system SWIs. Its most common use is to play sequences of notes, and a simplified interface is provided for this purpose.

A beat counter is used which is reset every time it reaches the end of a bar. Both its tempo and the number of beats to the bar can be programmed.

You may replace this module, although it is unlikely to be necessary.

Voice Generators

Voice Generators generate and output sound samples to the DMA buffer on receiving a Fill Request from the Channel Handler. Typical algorithms that might be used to synthesise a sound sample include calculation, lookup of filtered wavetables, or frequency modulation. A Voice Generator will normally allow multiple channels to be attached.

An interface exists for you to add custom Voice Generators, expanding the range of available sounds. The demands made on processor bandwidth by synthesis algorithms are high, especially for complex sounds, so you must write them with great care.

Technical details

DMA Handler

The DMA Handler manages the hardware used by the Sound system. Two (or more) physical buffers in main memory are used. These are accessed using four registers in the sound DMA Address Generator (DAG) within the Memory Controller chip:

- The DAG *sound pointer* points to the byte of sound to be output
- The *current end* register points to the end of the DMA buffer
- The *next start/end* register pair point to the most recently filled buffer.

The sound pointer is incremented every time a byte is read by the video controller for output. When it reaches the end of the current buffer the memory controller switches buffers: the sound pointer and buffer end registers are set to the values stored in the next start and next end registers respectively. An interrupt is then issued by the I/O controller indicating the buffers have switched, and the DMA handler is entered.

The DMA Handler calls the Channel Handler with a Fill request, asking that the next buffer be filled. (See below for details of the Channel Handler.) If this fill is completed, control returns to the DMA Handler and it makes the next start and next end registers point to the buffer just filled. If the fill is not completed then the next registers are not altered, and so the same buffer of sound will be repeated, causing an audible discontinuity.

Configuring the Sound system

The rest of this section outlines the factors that you must consider if you choose to reconfigure the Sound system.

Terminology used

- The *output period* is the time between each output of a byte.
- The *sample period* is the time between each output for a given channel.
- The *buffer period* is the time to output an entire buffer.

There are corresponding rates for each of the above.

- The *sample length* is the number of bytes in the buffer per channel.
- The *buffer length* is the total number of bytes in the buffer.

DMA Buffer period

A short buffer period is desirable to minimise the size of the buffer and to give high resolution to the length of notes; a long buffer period is desirable to decrease the frequency and number of interrupts issued to the processor. A period of approximately one centisecond is chosen as a default value, although this can be changed, for example to replay lengthy blocks of sampled speech from a disc.

Sample rate: maximum

A high sample rate will give the best sound quality. If too high a rate is sought then DMA request conflicts will occur, especially when high bandwidths are also required from the Video Controller by high resolution screen modes. To avoid such contention the output period must not be less than $4\mu\text{s}$. Outputting a byte to one of eight channels every $4\mu\text{s}$ results in a sample period of $32\mu\text{s}$, which gives a maximum sample rate of 31.25kHz.

Sample rate: default

The clock for the Sound system is derived from the system clock for the video controller, which is then divided by a multiple of 24. Current ARM based computers use a VIDC system clock of 24MHz; however, 20MHz and 28MHz clocks are also supported. The default output period is the shortest one that can be derived from all three clocks, thus ensuring that speech and music can be produced at the same pitch on any likely future hardware. This is $6\mu\text{s}$, obtained as follows:

- 20MHz clock divided by 120 (5×24)
- 24MHz clock divided by 144 (6×24)
- 28MHz clock divided by 168 (7×24)

Outputting a byte to one of eight channels every $6\mu\text{s}$ results in a sample period of $48\mu\text{s}$, which gives a default sample rate of 20.833kHz.

Buffer length

The DMA buffer length depends on the number of channels, the sample rate, and the buffer period. It must also be a multiple of 4 words. Using the defaults outlined above, the lengths shown in the middle two columns of the following table are the closest alternatives:

Buffer lengths for one centisecond sample, at sample rate of 20.833 kHz:

	Buffer length		Output period
1 channel	208 bytes	224 bytes	48 μ s
2 channels	416 bytes	448 bytes	24 μ s
4 channels	832 bytes	896 bytes	12 μ s
8 channels	1664 bytes	1792 bytes	6 μ s
Buffer period	0.9984cs	1.0752cs	
Interrupt rate	100.16Hz	93.01Hz	
Bytes per channel	&D0	&E0	

The system default buffer period is chosen as 0.9984 centi-seconds, thus the sample length is 208 bytes, or 52 words (13 DMA quad-word cycles). The buffer length is a multiple of this, depending on how many channels are used.

The sound DMA system systematically outputs bytes at the programmed sample rate; each (16-byte) load of DMA data from memory is synchronised to the first stereo image position. Each byte must be stored as an eight bit signed logarithm, ready for direct output to the VIDC chip:

Multiple channel operation is possible with two, four or eight channels; in this case the data bytes for each channel must be interleaved throughout the DMA buffer at two, four or eight byte intervals. When output the channels are multiplexed into what is effectively one half, one quarter or one eighth of the sample period, so the signal level per channel is scaled down by the same amount. Thus the signal level per channel is scaled, depending on the number of channels; but the overall signal level remains the same for all multi-channel modes.

DMA Buffer format

Showing the interleaving schematically:

Single channel format:

0	byte 0 chan 1	byte 1 chan 1	byte 2 chan 1	byte 3 chan 1	byte 4 chan 1	byte 5 chan 1	byte 6 chan 1	byte 7 chan 1
+8	byte 8 chan 1	byte 9 chan 1	byte 10 chan 1	byte 11 chan 1	byte 12 chan 1	byte 13 chan 1	etc...	

Output rate = 20 kHz

Image registers 0 - 7 programmed identically

Two channel format:

0	byte 0 chan 1	byte 0 chan 2	byte 1 chan 1	byte 1 chan 2	byte 2 chan 1	byte 2 chan 2	byte 3 chan 1	byte 3 chan 2
+8	byte 4 chan 1	byte 4 chan 2	byte 5 chan 1	byte 5 chan 2	byte 6 chan 1	byte 6 chan 2	etc...	

Output rate = 40 kHz

Image registers 0+2+4+8 and 1+3+5+7 programmed per channel

Four channel format:

0	byte 0 chan 1	byte 0 chan 2	byte 0 chan 3	byte 0 chan 4	byte 1 chan 1	byte 1 chan 2	byte 1 chan 3	byte 1 chan 4
+8	byte 2 chan 1	byte 2 chan 2	byte 2 chan 3	byte 2 chan 4	byte 3 chan 1	byte 3 chan 2	etc...	

Output rate = 80 kHz

Image registers 0+4, 1+5, 2+6 and 3+7 programmed per channel

Eight channel format:

0	byte 0 chan 1	byte 0 chan 2	byte 0 chan 3	byte 0 chan 4	byte 0 chan 5	byte 0 chan 6	byte 0 chan 7	byte 0 chan 8
+8	byte 1 chan 1	byte 1 chan 2	byte 1 chan 3	byte 1 chan 4	byte 1 chan 5	byte 1 chan 6	etc...	

Output rate = 160 kHz

Image registers programmed individually.

The Channel Handler manages the interleaving for you by passing the correct start address and increment to the Voice Generator attached to each channel.

Channel Handler

The Channel Handler registers itself with the DMA Handler by passing its address using `Sound_Configure`. At this address there must be a standard header:

Channel Handler

Offset	Value
0	pointer to fill code
4	pointer to overrun fixup code
8	pointer to linear-to-log table
12	pointer to log-scale table

The fill code handles fill requests from the DMA Handler. The Channel Handler translates the fill request to a series of calls to the Voice Generators, passing the required buffer offsets so that data from all channels correctly interleaves. Any unused channels within the buffer are set to zero by the Channel Handler so they are silent.

The overrun fixup code deals with channels that are not successfully filled within a single buffer period and hence repeat the same DMA buffer. This feature is no longer supported in RISC OS and the Channel Handler simply returns. (In the Arthur OS the offending channel was marked as overrun, the previous Channel Handler was aborted, and a new buffer fill initiated.)

The pointer to the linear-to-log table holds the address of the base of an 8 kbyte table which maps 32-bit signed integers directly to 8-bit signed volume-scaled logarithms in a suitable format for output to the VIXC chip.

Sound Channel Control Block (SCCB)

The pointer to the log-scale table holds the address of a 256-byte table which scales the amplitude of VIDC-format 8-bit signed logarithms from their maximum range down to a value scaled to the volume setting. Voice Generators should use this table to adjust their overall volume.

The Channel Handler maintains a 256 byte Sound Channel Control Block (SCCB) for each channel. An SCCB contains parameters and flags used by Voice Generators, and an extension area for programmers to pass any essential further data. Such an extension must be well documented, and used with care, as it will lead to Voice Generators that are no longer wholly compatible with each other.

The 9 initial words hold values that are normally stored in R0-R8 inclusive. They are saved to the SCCB using the instruction `LDMIA R9, {R0-R8}`

Offset	Value
0	gate bit + channel amplitude (7-bit log)
1	index to voice table
2	instance no. for attached voice
3	control/status bit flags
4	phase acc pitch oscillator
8	phase acc timbre oscillator
12	no. of buffer fills left to do (counter)
16	(normally working R4)
20	(normally working R5)
24	(normally working R6)
28	(normally working R7)
32	(normally working R8)
36 - 63	reserved for use by Acorn (28 bytes)
64 - 255	available for users

The flag byte indicates the state of the voice attached to the channel, and may be used for allocating voices in a polyphonic manner. Each time a Voice Generator completes a buffer fill and returns to the Channel Handler it returns an updated value for the Flags field in R0.

It is the responsibility of the Channel Handler to store the returned flag byte, and to update the other fields of each SCCB as necessary.

Note - In the Arthur OS, the flag byte was also used to detect channels that had overrun. If any were found then a call was made indirected through the fix up pointer (see above).

Voice Table

The Channel Handler uses a voice table recording the names of voices installed in the 32 available voice slots. It is always accessed through the SWI calls provided, and so its format is not defined.

Scheduler

Header

The Scheduler registers itself with the DMA Handler by passing its address using Sound_Configure. At this address there must be a pointer to the code for the Scheduler.

Use

Although the Scheduler is principally designed for queuing sound commands it can be used to issue other SWIs. Thus it could be used to control, for example, an external instrument interface (such as a Musical Instrument Digital Interface (MIDI) expansion podule), or a screen-based music editor with real-time score replay.

Extreme care must be used with the Scheduler, as it has limitations. R2 - R7 are always cleared when the SWI is issued, and the error-returning form ('X' form) of the SWI is forced. Return parameters are discarded. If pointers are to be passed in R0 or R1 then the data they address **must** be preserved until the SWI is called. If a SWI will not work within these limitations it must not be called by the Scheduler.

The Scheduler implements the queue as a circular chain of records. A stack listing the free slots is also kept. The number of free slots varies not only according to how many events are queued, but also to how the events are 'clustered'.

The queue is always accessed through the SWI calls provided, and so its precise format is not defined.

Event dispatcher

Every centisecond the beat counter is advanced according to the tempo value, and any events that fall within the period are activated in strict queuing order. Voice and parameter change events are processed and the SCCB for each Voice Generator updated as necessary by the Channel Handler, before fill requests are issued to the relevant Voice Generators.

Voice Generators

A Voice Generator is added to the Sound system by issuing a Sound_Install call, which passes its address to the Channel Handler. At this address there must be a standard header:

Header

Offset	Contents
0	B FillCode
4	B UpdateCode
8	B StartCode
12	B ReleaseCode
16	B Instantiate
20	B DeInstantiate
24	LDMFD R13!, {pc}
28	Offset from start of header to voice name

The Fill, Update, GateOn and GateOff entries provide services to fill the DMA buffer at different stages of a note, as detailed below.

The Instantiate and Free entries provide facilities to attach or detach the Voice Generator to or from a channel, as detailed below.

The Install entry was originally to be called when a Voice Generator was initialised. Since Voice Generators are now implemented as Relocatable Modules, which offer exactly this service in the form of the Initialisation entry point, this field is not supported and simply returns to the caller.

The voice name is used by the Channel Handler voice table. It should be both concise and descriptive. The offset must be positive relative – that is, the voice name must be after the header.

Buffer filling: entry conditions

A fill request to a Voice Generator is made by the Channel Handler using one of the four buffer fill entry points. The registers are allocated as follows:

Register	Function
R6	negative if configuration of Channel Handler changed
R7	channel number
R8	sample period in μ s
R9	pointer to SCCB (Sound Channel Control Block)
R10	pointer to end of DMA buffer
R11	increment to use when writing to DMA buffer
R12	pointer to (start of DMA buffer + interleaf offset)
R13	stack (Return address is on top of stack)
R14	do not use

Further parameters are available in the SCCB for that channel, which is addressed by R9. See the Channel Handler description for details. The usage of the parameters depends on which of the four entry points is called.

The ARM is in IRQ mode with interrupts enabled.

Buffer filling: routine conditions

The routine must fill the buffer with 8 bit signed logarithms in the correct format for direct output to the VIDC chip:

The ARM is in IRQ mode with interrupts enabled. They must remain enabled to ensure that system devices do not have a lengthy wait to be serviced. The code for a Voice Generator must therefore be re-entrant, and R14 must not be used as a subroutine link register, since an interrupt will corrupt it. Sufficient IRQ stack depth must be maintained for system IRQ handling. You can enter SVC mode if you wish.

Buffer filling: exit conditions

When a Voice Generator has completed a buffer fill it sets a flag byte in R0, and returns to the Channel Handler using `LDMFD R13!, {PC}`. The flag byte shows the status of each channel, and is used to prioritise fill requests to the Voice Generators.



Bit	Meaning
Q	Quiet (GateOff flag)
K	Kill pending (GateOn flag)
I	Initialise pending (Update flag)
F	Fill pending
A	Active (normal Fill in progress)
V	oVerrun flag (no longer supported)
F2, F1	2-bit Flush pending counter

Entry points for buffer filling

There are four different entry points for buffer filling, which are used at the different stages of a note. It is the responsibility of the Channel Handler to determine which Voice Generator to call, which entry should be used, and to update the SCCB as necessary when these calls return.

GateOn entry

The GateOn entry is used whenever a sound command is issued that requires a new envelope. Normally any previous synthesis is aborted and the algorithm restarted.

On exit the A bit (bit 3) of the flag byte is set.

Update entry

The Update entry is used whenever a sound command is issued that requires a smooth change, without a new envelope (using extended amplitudes `&180` to `&1FF` in the *Sound command for example). Normally the previous algorithm is continued, with only the amplitude, pitch and duration parameters supplied by the SCCB updated.

On exit the A bit (bit 3) of the flag byte is returned unless the voice is to stop sounding; for example if the envelope has decayed to zero amplitude. In these cases the F2 bit (bit 1) is set, and the Channel Handler will automatically flush out the next two DMA buffers, before becoming dormant.

Fill entry

The Fill entry is used when the current sound is to continue, and no new command has been issued.

On exit it is normal to return the same flags as for the Update entry.

GateOff entry

The GateOff entry is used to finish synthesising a sound. Simple voices may stop immediately, which is liable to cause an audible 'click'; more refined algorithms might gradually release the note over a number of buffer periods. A GateOff entry may be immediately followed by a GateOn entry.

On exit the F2 bit (bit 1) is set if the voice is to stop sounding, or the A bit (bit 3) is set if the voice is still being released.

Voice instantiation

Two entry points are provided to attach or detach a voice generator and a sound channel. On entry the ARM is in Supervisor mode, and the registers are allocated as follows:

Register	Function
R0	physical Channel number -1 (0 to 7)
R14	usable

The return address is on top of the stack. All other registers must be preserved by the routines, which must exit using `LDMFD R13!, {pc}`

R0 is preserved if the call was successful, else it is altered.

Instantiate entry

The Instantiate entry is called to inform the Voice Generator of a request to attach a channel to it. Each channel attached is likely to need some private workspace. A Voice Generator should ideally be able to support eight channels. The request can either be accepted (R0 preserved on exit), or rejected (R0 altered on exit).

Free entry

The usual reason for rejection is that an algorithm is slow and is already filling as many channels as it can within each buffer period: for example very complex algorithms, or ones that read long samples off disc.

The Free entry is called to inform the Voice Generator of a request to detach a channel from it. The call **must** release the channel and preserve all registers.

Sound_Configure (SWI &40140)

Configures the Sound system

On entry

R0 = no. of channels, rounded up to 1,2,4 or 8
R1 = sample size (in bytes per channel – default 208)
R2 = sample period (in μ s per channel – default 48)
R3 = pointer to Channel Handler (normally 0 to preserve system Handler)
R4 = pointer to Scheduler (normally 0 to preserve system Scheduler)

On exit

R0 - R4 = previous values

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt is used to configure the number of sound channels, the sample period and the sample size. It can also be used by specialised applications to replace the default Channel Handler and Scheduler.

All current settings may be read by using zero input parameters.

The actual values programmed are subject to the limitations outlined earlier.

Related SWIs

None

Related vectors

None

Sound_Enable (SWI &40141)

	Enables or disables the Sound system
On entry	R0 = new state: 0 for no change (read state) 1 for OFF 2 for ON
On exit	R0 = previous state 0 for OFF 1 for closedown imminent 2 for closedown in progress 3 for active ON
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This software interrupt is used to enable or disable all Sound interrupts and DMA activity. This guarantees to inhibit all Sound system bandwidth consumption once a successful disable has been completed.
Related SWIs	Sound_Speaker (SWI &40143), Sound_Volume (SWI &40180)
Related vectors	None

Sound_Stereo (SWI &40142)

	Sets the stereo position of a channel
On entry	R0 = channel (C) to program R1 = image position: 0 is centre 127 for max right -127 for max left -128 for no change (read state)
On exit	R0 preserved R1 = previous image position, or -128 if R0 = 8 on entry
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	For N physical channels enabled, this call will program stereo registers C, C+N, C+2N... up to stereo register 8. For example, if two channels are currently in use, and channel 1 is programmed, channels 3, 5 and 7 are also programmed; if channel 3 is programmed, channels 5 and 7 are also programmed, but not channel 1. This Software call only updates RAM copies of the stereo image registers and the new positions, in fact, take effect on the next sound buffer interrupt. IRQ code can call this SWI directly for scheduled image movement.
Related SWIs	None
Related vectors	None

Sound_Speaker (SWI &40143)

	Enables or disables the speaker(s)
On entry	R0 = new state: 0 for no change (read state) 1 for OFF 2 for ON
On exit	R0 = previous state 1 for OFF 2 for ON
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This software interrupt enables/disables the monophonic or stereophonic mixed signal(s) to the internal loudspeaker amplifier(s). It has no effect on the external stereo headphone/amplifier output. This SWI disables the speaker(s) by muting the signal; you may still be able to hear a very low level of sound.
Related SWIs	Sound_Enable (SWI &40141), Sound_Volume (SWI &40180)
Related vectors	None

Sound_Volume (SWI &40180)

	Sets the overall volume of the Sound system
On entry	R0 = sound volume (1 - 127) (0 to inspect last setting)
On exit	R0 = previous volume
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	<p>This call sets the maximum overall volume of the Sound system. A change of 16 in the volume will halve or double the volume. The command scales the internal lookup tables that Voice Generators use to set their volume; some custom Voice Generators may ignore these tables and so will be unaffected.</p> <p>A large amount of calculation is involved in this apparently trivial call. It should be used sparingly to limit the overall volume; the volume of each channel should then be set individually.</p>
Related SWIs	Sound_Enable (SWI &40141), Sound_Speaker (SWI &40143)
Related vectors	None

Sound_SoundLog (SWI &40181)

	Converts a signed integer to a signed logarithm, scaling it by volume
On entry	R0 = 32-bit signed integer
On exit	R0 = 8-bit signed volume-scaled logarithm
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call maps a 32-bit signed integer to an 8 bit signed logarithm in VIDC format. The result is scaled according to the current volume setting. Table lookup is used for efficiency.
Related SWIs	Sound_LogScale (SWI &40182)
Related vectors	None

Sound_LogScale (SWI &40182)

Scales a signed logarithm by the current volume setting

On entry

R0 = 8-bit signed logarithm

On exit

R0 = 8-bit signed volume-scaled logarithm

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt maps an 8-bit signed logarithm in VIDC format to one scaled according to the current volume setting. Table lookup is used for efficiency.

Related SWIs

Sound_SoundLog (SWI &40181)

Related vectors

None

Sound_InstallVoice (SWI &40183)

	Adds a voice to the Sound system
On entry	R0 = pointer to Voice Generator (0 for don't change) R1 = voice slot specified (0 for install in next free slot, else 1 - 32)
On exit	R0 = pointer to name of previous voice (or null terminated error string) R1 = voice number allocated (0 for FAIL to install)
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	<p>This software interrupt is used by Voice Modules or Libraries to add a Voice Generator to the table of available voices. If an error occurs, this SWI does not set V in the usual manner. Instead R1 is zero on exit, and R0 points directly to a null-terminated error string.</p> <p>Alternatively, the table of installed voices may be read by setting R0 to 0, and R1 to the slot to examine. If the slot is unused RISC OS gives a null pointer. (The Arthur OS gave a pointer to the string '*** No Voice'.)</p>
Related SWIs	Sound_RemoveVoice (SWI &40184)
Related vectors	None

Sound_RemoveVoice (SWI &40184)

Removes a voice from the Sound system

On entry

R1 = voice slot to remove (1 - 32)

On exit

R0 = pointer to name of previous voice (or error message)

R1 is voice number de-allocated (0 for FAIL)

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt is used when Voice Modules or Libraries are to be removed from the system. It notifies the Channel Handler that a RAM-resident Voice Generator is being removed. If an error occurs, this SWI does not set V in the usual manner. Instead R1 is zero on exit, and R0 points directly to a null-terminated error string.

This call must also be issued before the Relocatable Module Area is Tidied, since the module contains absolute pointers to Voice Generators that are likely to exist in the RMA.

Related SWIs

Sound_InstallVoice (SWI &40183)

Related vectors

None

Sound_AttachVoice (SWI &40185)

Attaches a voice to a channel

On entry

R0 = channel number (1 - 8)

R1 = voice slot to attach (0 to detach and mute channel)

On exit

R0 preserved (or 0 if illegal channel number)

R1 = previous voice number (or 0 if not previously attached)

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call attaches a voice with a given slot number to a channel. The previous voice is shut down and the new voice is reset.

Different algorithms have different internal state representations so it is not possible to swap Voice Generators in mid-sound.

Related SWIs

Sound_AttachNamedVoice (SWI &4018A)

Related vectors

None

Sound_ControlPacked (SWI &40186)

Makes an immediate sound

On entry

R0 is AAAACCCC Amp/Channel
R1 is DDDDPPPP Duration/Pitch

On exit

R0,R1 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call is identical to Sound_Control (SWI &40189), but the parameters are packed 16-bit at a time into low R0, high R0, low R1, high R1 respectively. It is provided for BBC compatibility and for the use of the Scheduler. The Sound_Control call should be used in preference where possible.

Related SWIs

Sound_Control (SWI &40189)

Related vectors

None

Sound_Tuning (SWI &40187)

	Sets the tuning for the Sound system
On entry	R0 = new tuning value (or 0 for no change)
On exit	R0 = previous tuning value
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call sets the tuning for the Sound system in units of 1/4096 of an octave. The command *Tuning 0 may be used to restore the default tuning.
Related SWIs	None
Related vectors	None

Sound_Pitch (SWI &40188)

Converts a pitch to internal format (a phase accumulator value)

On entry

R0 = 15-bit pitch value:
bits 14 - 12 are a 3-bit octave number
bits 11 - 0 are a 12-bit fraction of an octave (in units of 1/4096 octave)

On exit

R0 = 32-bit phase accumulator value, or preserved if R0 &8000 on entry

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This software interrupt maps a 15-bit pitch to an internal format pitch value (suitable for the standard voice phase accumulator oscillator).

Related SWIs

None

Related vectors

None

Sound_Control (SWI &40189)

	Makes an immediate sound
On entry	R0 = channel number (1 - 8) R1 = amplitude: &FFF1 - &FFFF and 0 for BBC emulation amplitude (0 to -15) &0001 - &000F BBC envelope not emulated &0100 - &01FF for full amplitude/gate control: bit 7 is 0 for gate ON/OFF 1 for smooth update (gate not retriggered) bits 6 - 0 are 7-bit logarithm of amplitude R2 = pitch &0000 - &00FF for BBC emulation pitch &0100 - &7FFF for enhanced pitch control: bits 14 - 12 = 3-bit octave bits 11 - 0 = 12-bit fractional part of octave (&4000 is nominally Middle C) &8000 + n 'n' (in range 0 - &7FFF) is phase accumulator increment R3 = duration &0001 - &00FE for BBC emulation in 5 centisecond periods &00FF for BBC emulation 'infinite' time (converted to &F0000000) > &00FF for duration in 5 centisecond periods.
On exit	R0 - R3 preserved
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call allows real-time control of a specified Sound Channel. The parameters are immediately updated and take effect on the next buffer fill.

Gate on and off correspond to the start and end of a note and of its envelope (if implemented). 'Smooth' update occurs when note parameters are changed without restarting the note or its envelope – for example when the pitch is changed to achieve a glissando effect..

If any of the parameters are invalid the call does not generate an error; instead it returns without performing any operation.

Related SWIs

Sound_ControlPacked (SWI &40186)

Related vectors

None

Sound_AttachNamedVoice (SWI &4018A)

Attaches a named voice to a channel

On entry

R0 = channel number (1 - 8)

R1 = pointer to voice name (ASCII string, null terminated)

On exit

R0 is preserved, or 0 for fail

R1 is preserved

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call attaches a named voice to a channel. If no exact match for the name is found then an error is generated and the old voice (if any) remains attached. If a match is found then the previous voice is shut down and the new voice is reset.

Different algorithms have different internal state representations so it is not possible to swap Voice Generators in mid-sound.

Related SWIs

Sound_AttachVoice (SWI &40185)

Related vectors

None

Sound_ReadControlBlock (SWI &4018B)

Reads a value from the Sound Channel Control Block

On entry

R0 = channel number (1 - 8)
R1 = offset to read from (0 - 255)

On exit

R0 preserved (or 0 if fail, invalid channel, or invalid read offset)
R1 preserved
R2 = 32-bit word read (if R0 non-zero on exit)

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call reads 32-bit data values from the Sound Channel Control Block (SCCB) for the designated channel. This call can be used to read parameters not catered for in the Sound_Control calls returned by Voice Generators, using an area of the SCCB reserved for the programmer.

Related SWIs

Sound_WriteControlBlock (SWI &4018C)

Related vectors

None

Sound_WriteControlBlock (SWI &4018C)

	Writes a value to the Sound Channel Control Block
On entry	R0 = channel number (1 - 8) R1 = offset to write to (0 - 255) R2 = 32-bit word to write
On exit	R0 preserved (or 0 if fail, invalid channel, or invalid write offset) R1 preserved R2 = previous 32-bit word (if R0 non-zero on exit)
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call writes 32-bit data values to the Sound Channel Control Block (SCCB) for the designated channel. This call can be used to pass parameters not catered for in the Sound_Control calls to Voice Generators, using an area of the SCCB reserved for the programmer.
Related SWIs	Sound_ReadControlBlock (SWI &4018B)
Related vectors	None

Sound_QInit (SWI &401C0)

	Initialises the Scheduler's event queue
On entry	No parameters passed in registers
On exit	R0 = 0, indicating success
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call flushes out all events currently scheduled and re-initialises the event queue. The tempo is set to the default, the beat counter is reset and disabled, and the bar length set to zero.
Related SWIs	None
Related vectors	None

Sound_QSchedule (SWI &401C1)

	Schedules a sound SWI on the event queue
On entry	R0 = schedule period -1 to synchronise with the previously scheduled event -2 for immediate scheduling R1 = 0 to schedule a Sound_ControlPacked call, or SWI code to schedule (of the form &xF000000 + SWI no.) R2 = SWI parameter to be passed in R0 R3 = SWI parameter to be passed in R1
On exit	R0 = 0 for successfully queued R0 < 0 for failure (queue full)
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call schedules a sound SWI call. If the beat counter is enabled the schedule period is measured from the last start of a bar, otherwise it is measured from the time the call is made. A schedule time of -1 forces the new event to be queued for activation concurrently with the previously scheduled one. The event is typically a Sound_ControlPacked type call, although any other sound SWI may be scheduled. There are limitations: R2 - R7 are always cleared, and any return parameters are discarded. If pointers are to be passed in R0 or R1 then any associated data must still remain when the SWI is called (the workspace involved must not have been reused, the Window Manager must not have paged it out, and so on).
Related SWIs	Sound_QFree (SWI &401C3)
Related vectors	None

Sound_QRemove (SWI &401C2)

This SWI call is for use by the Scheduler only. You must not use it in your own code.

Sound_QFree (SWI &401C3)

Returns minimum number of free slots in the event queue

On entry

No parameters passed in registers

On exit

R0 = no. of guaranteed slots free

R0 < 0 indicates over worst case limit, but may still be free slots

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns the minimum number of slots guaranteed free. The calculation assumes the worst case of data structure overheads that could occur, so it is likely that more slots can in fact be used. If this guaranteed free slot count is exceeded this call will return negative values, and the return status of QSchedule must be carefully monitored to observe when overflow occurs.

Related SWIs

Sound_QSchedule (SWI &401C1)

Related vectors

None

Sound_QSDispatch (SWI &401C4)

This SWI call is for use by the Scheduler only. You must not use it in your own code.

Sound_QTempo (SWI &401C5)

	Sets the tempo for the Scheduler
On entry	R0 = new tempo (or 0 for no change)
On exit	R0 = previous tempo value
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	<p>This command sets the tempo for the Scheduler. The default tempo is &1000, which corresponds to one beat per centisecond; doubling the value doubles the tempo (ie &2000 gives two beats per centisecond), while halving the value halves the tempo (ie &800 gives half a beat per centisecond).</p> <p>The parameter can be thought of as a hexadecimal fractional number, where the three least significant digits are the fractional part.</p>
Related SWIs	Sound_QBeat (SWI &401C6)
Related vectors	None

Sound_QBeat (SWI &401C6)

Sets or reads the beat counter or bar length

On entry

R0 = 0 to return current beat number
R0 = -1 to return current bar length
R0 < -1 to disable beat counter and set bar length 0
R0 = +N to enable beat counter with bar length N (counts 0 to N-1)

On exit

R0 = current beat number (R0 = 0 on entry), otherwise the previous bar length.

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

The simplest use of this call is to read either the current value of the beat counter or the current bar length.

When the beat counter is disabled both it and the bar length are reset to zero. All scheduling occurs relative to the time the scheduling call is issued.

When the beat counter is enabled it is reset to zero. It then increments, resetting every time it reaches the programmed bar length (N-1). Scheduling using QSchedule then occurs relative to the last bar reset; however, scheduling using *QSound is still relative to the time the command is issued.

Related SWIs

Sound_QTempo (SWI &401C5)

Related vectors

None

Sound_QInterface (SWI &401C7)

This SWI call is for use by the Scheduler only. You must not use it in your own code.

* Commands

*Audio

Turns the Sound system on or off

Syntax

*Audio ON | OFF

Parameters

None

Use

Turning Audio Off silences the Sound system completely, stopping all Sound interrupts and DMA activity. Turning Audio back on restores the Sound DMA and interrupt system to the state it was in immediately prior to turn-off. All Channel Handler and Scheduler activity is effectively frozen during the time the Audio system is off, but software interrupts are still permitted, even if no sound results.

Example

*Audio ON

Related commands

*Speaker, *Volume

Related SWIs

Sound_Enable (SWI &40141)

Related vectors

None

*ChannelVoice

Attaches a voice to a channel

Syntax

```
*ChannelVoice <channel> <voice slot>|<voice name>
```

Parameters

```
<channel>           from 1 to 8  
<voice slot>       from 1 to 16, as given by *Voices  
<voice name>      name, as given by *Voices
```

Use

This command attaches a voice to a channel. The voice slot is that given by the *Voices command, and depends on the order in which voices were loaded. It is preferable to use the voice names; note that these are case sensitive. Alternatively, the channel can be muted by passing a voice slot of 0.

Example

```
*ChannelVoice 1 StringLib-Pluck
```

Related commands

```
*Stereo, *Voices
```

Related SWIs

```
Sound_AttachVoice (SWI &40185),  
Sound_AttachNamedVoice (SWI &4018A)
```

Related vectors

```
None
```

*Configure SoundDefault

Sets the default volume and voice

```
*Configure SoundDefault <speaker> <volume> <voice>
```

Parameters

<speaker>	0 or 1
<volume>	from 0 to 7
<voice>	from 1 to 16, as given by *Voices

Use

This configure option sets the sound parameters kept in CMOS RAM for use after power-on. They specify the built-in loudspeaker(s) as on (one) or off (zero), the relative default volume preferred at start up, and the voice slot one wishes to attach to channel 1 (the default system Bell channel).

Example

```
*Configure SoundDefault 1 7 1
```

Related commands

None

Related SWIs

None

Related vectors

None

*QSound

Generates a sound after a given delay

Syntax

*QSound <channel> <amplitude> <pitch> <duration> <beats>

Parameters

- The channel (1 to 8) will only sound if at least that number of channels have been selected, and the channel has a voice attached.
- The amplitude can be expressed in two ways: the values 0 (silent) and &FFFF (almost silent) down to &FFF1 (loud) provide a linear scale; and the range &100 (silent) to &17F (loud) provides a logarithmic scale, where a change of 16 will halve or double the amplitude.
- The pitch can also be expressed in two ways: for the range 0 to 255, each unit represents a quarter of a semitone, with a value of 53 producing middle C; for the range 256 (&100) to 32767 (&7FFF) the bottom 12 bits give the fraction of an octave, and the top three bits the octave – middle C has the value 16384 (&4000).
- The duration is given in twentieths of a second and must lie in the range 0 to 32767 (&8000). A value of 255 (&FF) is special: the sound will be continuous, stopping only when the escape key is pressed.
- The beats occur at the rate set by *Tempo

Use

This command is identical in effect to issuing a *Sound command after the specified number of beats have occurred.

Example

```
*QSound 1 &FFF2 &5800 10 50
```

Related commands

*Sound, *Tempo

Related SWIs

Sound_QSchedule (SWI &401C1)

Related vectors

None

*Sound

Generates an immediate sound

Syntax

*Sound <channel> <amplitude> <pitch> <duration>

Parameters

- The channel (1 to 8) will only sound if at least that number of channels have been selected, and the channel has a voice attached.
- The amplitude can be expressed in two ways: the values 0 (silent) and &FFFF (almost silent) down to &FFF1 (loud) provide a linear scale; and the range &100 (silent) to &17F (loud) provides a logarithmic scale, where a change of 16 will halve or double the amplitude.
- The pitch can also be expressed in two ways: for the range 0 to 255, each unit represents a quarter of a semitone, with a value of 53 producing middle C; for the range 256 (&100) to 32767 (&7FFF) the bottom 12 bits give the fraction of an octave, and the top three bits the octave – middle C has the value 16384 (&4000).
- The duration is given in twentieths of a second and must lie in the range 0 to 32767 (&8000). A value of 255 (&FF) is special: the sound will be continuous, stopping only when the escape key is pressed.

Use

This command generates an immediate sound.

Example

```
*Sound 1 &FFF2 &5800 10
```

Related commands

*QSound

Related SWIs

Sound_ControlPacked (SWI &40186), Sound_Control (SWI &40189)

Related vectors

None

*Speaker

	Turns the loudspeaker on or off
Syntax	*Speaker ON OFF
Parameters	None
Use	<p>This command mutes the monophonic or stereophonic mixed signal(s) to the internal loudspeaker amplifier(s). It does not effect the external stereo headphone/amplifier output.</p> <p>You may still be able to hear a very low level of sound, as this command does not totally disable the speaker(s).</p>
Example	*Speaker OFF
Related commands	*Audio, *Volume
Related SWIs	Sound_Speaker (SWI &40143)
Related vectors	None

*Stereo

Sets the stereo image position of a sound channel.

Syntax

```
*Stereo <channel> <position>
```

Parameters

<channel> from 1 to 8

<position> from -127(full left) to +127(full right), 0 for centre

Use

This command sets the stereo image position of a sound channel.

Example

```
*Stereo 1 100
```

Related commands

*ChannelVoice, *Voices

Related SWIs

Sound_Stereo (SWI &40142)

Related vectors

None

*Tempo

Sets the tempo for the Scheduler

Syntax

*Tempo <tempo>

Parameters

<tempo> from 0 to &FFFF (default &1000)

Use

This command sets the tempo for the Scheduler. The default tempo is &1000, which corresponds to one beat per centisecond; doubling the value doubles the tempo (so &2000 gives two beats per centisecond), while halving the value halves the tempo (so &800 gives half a beat per centisecond).

Example

*Tempo &1200

Related commands

*QSound

Related SWIs

Sound_QTempo (SWI &401C5)

Related vectors

None

*Tuning

Alters the Sound system tuning

Syntax	*Tuning <relative change>
Parameters	<relative change> from -16383 to 16383 (0 resets the default tuning)
Use	This command alters the tuning for the Sound system. A value of zero resets the default tuning. Otherwise, the tuning is changed relative to its current value in units of 1/4096 of an octave.
Example	*Tuning 64
Related commands	None
Related SWIs	Sound_Tuning (SW1 &40187)
Related vectors	None

*Voices

	Lists the installed voice generators
Syntax	*Voices
Parameters	None
Use	This command lists the voice generators that are installed, and the channel(s) that each is attached to (if any). A voice can be attached to a channel even if that channel is not currently in use.
Example	*Voices
Related commands	*ChannelVoice, *Stereo
Related SWIs	Sound_InstallVoice (SWI &40183)
Related vectors	None

*Volume

Sets the maximum overall volume

Syntax

```
*Volume <vol>
```

Parameters

```
<vol>          from 1 to 127
```

Use

This command sets the maximum overall volume of the Sound system. A change of 16 in the volume will halve or double the volume. The command scales the internal lookup tables that Voice Generators use to set their volume; some custom Voice Generators may ignore these tables and so will be unaffected.

A large amount of calculation is involved in this apparently trivial command. It should be used sparingly to limit the overall volume; the volume of each channel should then be set individually.

Example

```
*Volume 127
```

Related commands

```
*Audio, *Configure SoundDefault, *Speaker
```

Related SWIs

```
Sound_Volume (SWI &40180)
```

Related vectors

```
None
```

Application notes

The most likely change to the Sound system is to add Voice Generators, thus providing an extra range of sounds. Each Voice Generator must conform to the specifications given earlier in the section entitled *Technical Details*, and those given below. The speed and efficiency of Voice Generator algorithms is paramount, and requires careful attention to coding; some suggested code fragments are given to help you.

Code will not run fast enough in ROM, so ROM templates or user code templates must be copied into the Relocatable Module Area where they will execute in fast sequential RAM. If the RMA is to be tidied, all installed voices must be removed using the `Sound_Remove` call, then reinstalled using the `Sound_Install` call.

Voice libraries are an efficient way of sharing common code and data areas; these must be built as Relocatable Modules which install sets of voices, preferably with some form of library name prefix.

Buffer filling algorithms

The Channel Handler sets up three registers (R12,11,10) which give the start address, increment and end address for correct filling with interleaved sound samples. The interleave increment has the value 1, 2, 4 or 8, and is equal to the number of channels. This code is an example of how these registers should be used:

```
.loop
    ...
    ... ; e.g. form VIDC format 8 bit signed log in Rs
    STRB Rs,[R12],R11 ; store, and bump ptr
    CMPS R12,R10 ; check for end
    BLT loop ; and loop until fill complete
```

The DMA buffer is always a multiple of 4 words (16 bytes) long, and word aligned. Loop overheads can therefore be cut down by using two byte store operations. A further improvement is possible if R11, the increment, is one; this implies that values are to be stored sequentially, so word stores may be used.

Example code fragments

The fundamental operations performed by nearly all voice generators involve Oscillators, Table lookup and Amplitude modulation. In addition, some algorithms (plucked string and drum in particular) require random bit generators. Simple in-line code fragments are briefly outlined for each of these.

Note: in all cases the aim is to produce the most efficient, and wherever possible highly sequential, ARM machine code; in most algorithms the aim must be to get as many working variables into registers as possible, and then adapt the synthesis algorithms wherever possible to use the high-speed barrel shifter to effect.

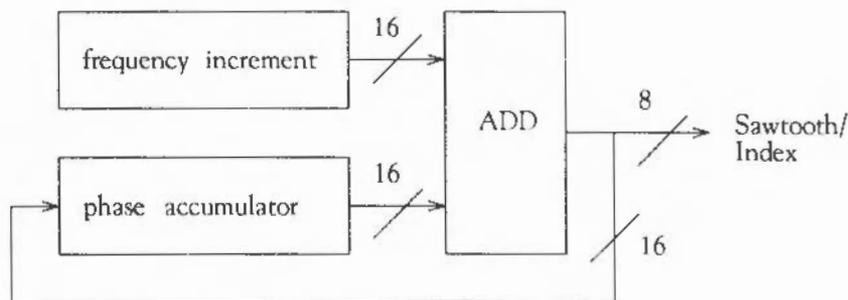
Oscillator coding

The accumulator-divider is the most useful type of oscillator for most voices. A frequency increment is added to a phase accumulator register and the high-order bits of the resulting phase provide the index to a wavetable. Alternatively, the top byte can be directly used as a sawtooth waveform.

The frequency of the oscillator is linearly related to the frequency increment. Vibrato effects can be obtained by modulating the frequency increment

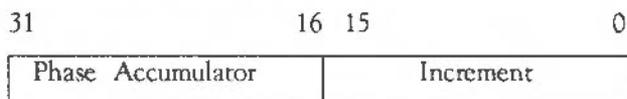
Sixteen-bit registers provide good audible frequency resolution, and are used in many digital hardware synthesizer products. The 32-bit register width of the ARM is ideally split 16/16 bits for phase/increment.

Schematically



Coding

Register field assignment: Rp



```
ADD Rp,Rp,Rp,LSL #16 ; phase accumulate
```

Changing parameters or the voice table being used is best done at or close to zero-crossing points, to avoid noise generation. If wavetables are arranged with zero-crossing aligned to the start and end of the table then it is simple to add a branch to appropriate code.

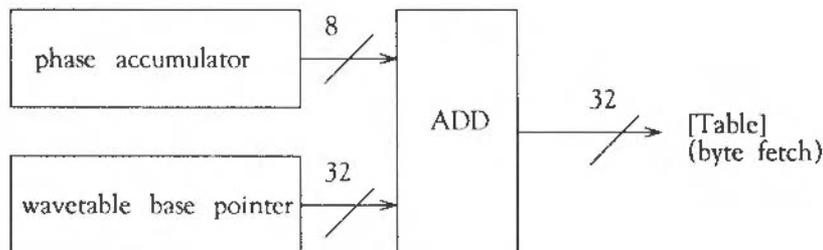
```
ADDS Rp,Rp,Rp,LSL #16 ; phase accumulate  
BCS Update ; only take branch if past zero crossing
```

Wavetable access coding

Normally fixed-length (256-byte or a larger power of two) wavetables are used by most voice generator modules. The high bits of the phase accumulator are added to a wavetable base pointer to access the sample byte within the table:

Schematically

For a 256-byte table:



Coding

```
LDRB Rs,[Rt,Rp,LSR #24]
```

where the most significant 8 bits of Rp contain the Phase index, Rt is the Table base pointer, and Rs is the register used to store the sample.

Amplitude modulation coding

The amplitude of the resultant byte may be altered for three reasons: firstly to scale for the overall volume setting, secondly to scale for the channel's volume setting, and lastly to provide enveloping.

Overall volume

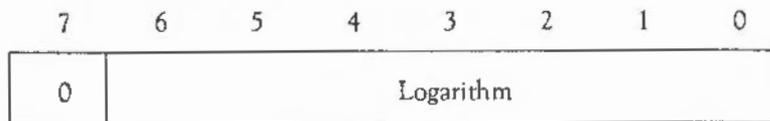
If the overall volume setting changes, then your Update entry point will be called. You can cope with the change in two ways. The first is to re-scale all the values in the wavetable, using the SWI calls SoundLog or LogScale. This has the advantage that buffer filling is faster as the values are already scaled, but has the disadvantage that the wavetables might be stored to a lower resolution resulting in increased noise levels.

The alternative is to re-scale the values between reading them from the wavetable and outputting them, as in the example voice given later. The reverse then applies: buffer filling is slower, but noise is reduced. This method is preferred, so long as the algorithm is still able to fill the buffer within the required period.

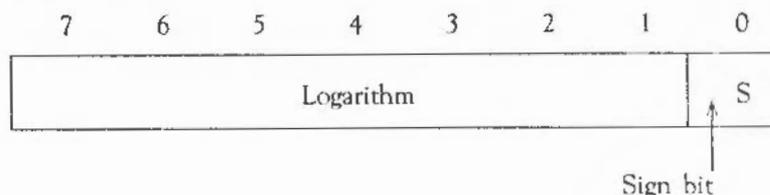
Channel volume

The channel's volume setting should be used by all well-behaved Voice Generators. The volume is passed to the Voice Generator by the Channel Handler in the SCCB, as a signed 8 bit logarithm, but in a different format to that used by the VIDC chip:

Amplitude Byte Data Format:



VIDC 8-bit sample format:



Coding

The coding is easiest if the values are treated as fractional quantities, and is then reduced to subtracting logarithms and checking for underflow:

Ra contains amplitude in range 0 to 127

Rs contains sample data in range -127 to +127 [sign bit LSB]

```
; do this each time Voice Generator is entered
RSB  Ra,Ra,#127          ; make attenuation factor

; do this inside loop, before each write to buffer
SUBS Rs,Rs,Ra,LSL #1    ; note shift to convert to VIDC format
MOVMI Rs,#0             ; correct for underflow
```

Note: The example voice shows how this can be combined with use of the volume-scaled lookup table to scale for both the overall and channel volume on each fill.

Envelope coding

Envelopes (if used) must be coded within the Voice Generator. A lookup table must be defined giving the envelope shape. This is then accessed in a similar manner to a wavetable, using the timbre phase accumulator passed in the SCCB. The sample byte is then scaled using this value, as shown above.

If you continue after a gate off, you must store your own copy of the volume, as any value in the SCCB will be overwritten.

Linear to logarithmic conversion

Algorithms which work with linear integer arithmetic may use the Channel Handler linear-log table directly to fill buffers efficiently. The table is 8 kbyte in length, to allow the full dynamic range of the VIDC sound digital to analogue converter to be utilised. The format is chosen to allow direct indexing using barrel-shifted 32-bit integer values. The values in the table are scaled according to the current volume setting.

Coding

```
; to access the lookup table pointer during initialisation:
MOV    R0,#0
MOV    R1,#0
MOV    R2,#0
MOV    R3,#0      ; get Channel Handler base
MOV    R4,#0
SWI    "XSound_Configure"
BVS    error_return
LDR    R8,[R3,#8] ; lin-to-log pointer

; in line buffer filling code:
; linear 32-bit value in R0
LDRB   R0,[R8,R0,LSR #19] ; lin -> log
STRB   R0,[R12],R11      ; output to DMA buffer
```

Random bit generator code

An efficient pseudo-random bit generator can be implemented using two internal registers. This provides noise which is necessary for some sounds, percussion in particular. One register is used as a multi-tap shift register, loaded with a seed value; the second is loaded with an XOR bit mask constant (&1D872B41). The sequence produced has a length of 4294967295. The random carry bit setting by the simple code fragment outlined below allows conditional execution on carry set (or cleared):

Coding

```
MOVS   R8,R8,LSL #1 ; set random carry
EORCS  R8,R8,R9
xxxCC  ; do this...
yyyCS  ; ...or alternately this
```

Example program

This program shows a complete Voice Generator. It builds a wavetable containing a sine wave at maximum amplitude. Scaling is performed when the table is read:

```
REM -> WaveVoice
:
DIM WaveTable% 255
DIM Code% 4095
:
SYS "Sound_Volume",127 TO UserVolume
FOR s%=0 TO 255
  SYS "Sound_SoundLog",&7FFFFFFF*SIN(2*PI*s%/256) TO WaveTable%?s%
NEXT s% : REM build samples at full volume
SYS "Sound_Volume",UserVolume TO UserVolume
REM and restore volume to value on entry
:
FOR C=0 TO 2 STEP 2
P%=Code%
[ OPT C
;*****
;* VOICE CO-ROUTINE CODE SEGMENT *
;*****
; On installation, point Channel Handler voice
; pointers to this voice control block
; (return address always on top of stack)
.VoiceBase
  B Fill
  B Fill ; update entry
  B GateOn
  B GateOff
  B Instance ; Instantiate entry
  LDMFD R13!,{PC} ; Free entry
  LDMFD R13!,{PC} ; Initialise
  EQU  VoiceName - VoiceBase
;
.VoiceName EQU "WaveVoice"
  EQU 0
  ALIGN
;*****
.LogAmpPtr EQU 0
.WaveBase EQU WaveTable%
;*****
.Instance ; any instance must use volume scaled log amp table
  STMPD R13!,{R0-R4} ; save registers
  MOV R0,#0
  MOV R1,#0
  MOV R2,#0
  MOV R3,#0
  MOV R4,#0
  SWI "XSound_Configure"
  LDRVC R0,{R3,#12} ; get address of volume scaled log amp table
  STRVC R0,LogAmpPtr ; and store
```

```

        STRVS    R0, [R13]          ; return error pointer
        LDMFD   R13!, {R0-R4, PC}  ; restore registers and return
;*****
;* VOICE BUFFER FILL ROUTINES      *
;*****
; on entry:
;   r0-r8 available
;   r9 is SoundChannelControlBlock pointer
;   r10 DMA buffer limit (+1)
;   r11 DMA buffer interleave increment
;   r12 DMA buffer base pointer
;   r13 Sound system Stack with return address and flags
;   on top (must LDMFD R13!, {...,pc}
; NO r14 - IRQs are enabled and r14 is not usable
.GateOn
        LDR     R0, WaveBase        ; wavetable base
        STR     R0, [R9, #16]      ; set up in SCCB as working register 5
        LDR     R0, LogAmpPtr      ; volume scaled log amp table
        STR     R0, [R9, #20]      ; set up as working register 6
;*****
.Fill
        LDMIA   R9, {R1-R5}        ; pick up working registers from SCCB
        AND     R1, R1, #&7F       ; mask R1 so only channel amplitude remains
; R1 is amp {0-127}      R2 is pitch phase acc
; R3 is timbre phase acc R4 is duration
; R5 is wavetable base  R6 is amp table base
; move sign bit -> VIDC format log
        LDRB   R1, [R6, R1, LSL #1] ; and lookup amp scaled to overall volume
        MOV    R1, R1, LSR #1      ; move sign bit back again
        RSB    R1, R1, #127        ; make attenuation factor
.FillLoop
        ADD    R2, R2, R2, LSL #16 ; advance waveform phase
        LDRB   R0, [R5, R2, LSR #24] ; get wave sample
        SUBS   R0, R0, R1, LSL #1  ; scale amplitude for overall & channel volumes
        MOVMI  R0, #0              ; and correct underflow
        STRB   R0, [R12], R11      ; generate output sample
        ADD    R2, R2, R2, LSL #16 ; repeated in line four times...
        LDRB   R0, [R5, R2, LSR #24]
        SUBS   R0, R0, R1, LSL #1
        MOVMI  R0, #0
        STRB   R0, [R12], R11
        ADD    R2, R2, R2, LSL #16
        LDRB   R0, [R5, R2, LSR #24]
        SUBS   R0, R0, R1, LSL #1
        MOVMI  R0, #0
        STRB   R0, [R12], R11      ; end of repeats...
        CMP    R12, R10           ; check for end of buffer fill
        BLT    FillLoop          ; loop if not

```

```

; check for end of note
SUBS   R4,R4,#1           ; decrement centisecc count
STMIB  R9,(R2-R5)        ; save registers to SCCB
MOVPL  R0,#%00001000    ; voice active if still duration left
MOVMI  R0,#%00000010    ; else force flush
LDMFD  R13!,{PC}        ; return to level 1
;*****
.GateOff
MOV    R0,#0
.FlushLoop
STRB   R0,[R12],R11     ; fill buffer with zeroes
STRB   R0,[R12],R11
STRB   R0,[R12],R11
STRB   R0,[R12],R11
CMP    R12,R10
BLT    FlushLoop
; CAUSE level 1 TO FLUSH once more
MOV    R0,#%00000001    ; set flag to flush one more buffer
LDMFD  R13!,{PC}        ; return to level 1
]
NEXT C
:
DIM OldVoice%(8)
SYS "Sound_InstallVoice",VoiceBase,0 TO a%,Voice%
FOR v%=1 TO 8
  SYS "Sound_AttachVoice",v%,0 TO z%,OldVoice%(v%)
  VOICE v%,"WaveVoice"
NEXT
:
ON ERROR PROCrestoreSound : END
:
VOICES 8
*voices
SOUND 1,€17F,53,10 :REM activate channel !!
PRINT'"any key to make a noise, <ESCAPE> to finish"
:
C%=1
REPEAT
  K%=INKEY(1)
  IF K%>0 THEN
    SOUND C%,€17F,K%,100
    C%+=1 : IF C%>8 THEN C%=1
  ENDIF
UNTIL 0
:
DEF PROCrestoreSound
ON ERROR OFF
REPORT:PRINT ERL
SYS "Sound_RemoveVoice",0,Voice%
FOR v%=1 TO 8
  SYS "Sound_AttachVoice",v%,OldVoice%(v%)
NEXT
VOICES 1

```

```
*voices  
PRINT''  
ENDPROC
```

WaveSynth

Introduction

WaveSynth is a module that provides:

- a voice generator which is used for the default system bell
- a SWI for its own internal use, that is used to load new wave tables.

For more information about the use of sound in RISC OS, refer to the chapter entitled *The Sound system*.

SWI Calls

WaveSynth_Load (SWI &40300)

Load new wave tables

Use

This software interrupt is for internal use only. It is used by the WaveSynth module to load new wave tables.

Expansion Cards

Introduction

Expansion Cards provide you with a way to add hardware to your RISC OS computer. They plug into slots provided in the computer, typically in the form of a backplane (these are an optional extra on some models).

This chapter gives details of the software that RISC OS provides to manage and communicate with expansion cards. It also gives details of what software and data needs to be provided by your expansion cards for RISC OS to communicate with them; in short, all you need to know to write the software for an expansion card.

What this chapter does not tell you is how to design the hardware. This is because:

- the range of hardware that can be added to a RISC OS computer is so large that we can't examine them all
- we don't have the space to describe every RISC OS computer that Acorn makes

Instead, you should see the further sources of information to which we refer you.

Overview

RISC OS computers can support internal slots for expansion cards. If you wish to add more cards than can be fitted to the supplied slots, you must use one of the slots to support an expansion card that buffers the signals on the expansion card bus before passing them on to external expansion cards.

Software

Expansion cards can have some or all of the following software included:

- an Expansion Card Identity, to give RISC OS information about the card
- Interrupt Status Pointers, to tell RISC OS where to look to find out if the card is generating interrupts
- a Chunk Directory, that defines what separate parts of the card's memory space are used for
- a Loader, to access paged memory held outside the card's address space
- a Chunk Directory to define what separate parts of the paged memory are used for.

A wide range of different types of code and data is supported by the Chunk Directories.

The use of the Loader and paged memory has been made as transparent to the end user as possible.

Technical Details

Expansion Card Identity

Each expansion card must have an *Expansion Card Identity* (or *ECId*) so that RISC OS can tell whether an expansion card is fitted in a backplane slot, and if so, identify it. The ECId may be:

- a simple ECId of only one byte (the low one of a word)
- an extended ECId of eight bytes, which may be followed by other information.

The ECId (whether extended or not) must appear at the bottom of the expansion card space immediately after a reset. However, it does not have to remain readable at all times, and so it can be in a paged address space so long as the expansion card is set to the page containing the ECId on reset.

The ECId is read by a synchronous read of address 0 of the expansion card space. You may only assume it is valid from immediately after a reset until when the expansion card driver is installed.

Expansion card identity space

The first 16 bytes of the expansion card identity space are always assumed to be byte-wide. If the ECId is included in a ROM which is 16 or 32 bits wide, then only the lowest byte in each half-word or word must be used for the first 16 (half) words.

If you use an extended ECId, you may specify the space after this as 8, 16 or 32 bits wide. When you access this space

- if you are using the 8 bit wide mode, you should use byte load and store instructions
- if you are writing using the 16 bit wide mode, you should use word store instructions, putting your half word in both the low and high half words of the register you use
- if you are reading using the 16 bit wide mode, you should use word load instructions, and ignore the upper half word returned
- if you are using the 32 bit wide mode, you should use word load and store instructions.

Synchronous cycles are used by the operating system to read and write any locations within this space (to simplify the design of synchronous expansion cards).

You should note however that there are currently some restrictions on the widths you can use. These are imposed both by current hardware and software:

- the I/O data bus is only 16 bits wide
- the current version of the RISC OS Expansion Card Manager only supports the 8 bit wide mode; future versions may support the wider modes.

Simple Expansion Card Identity

A simple ECId is one byte long. You should only use one for the very simplest of expansion cards, or temporarily during development. Most expansion cards should implement the extended ECId which eliminates the possibility of expansion card IDs clashing. A simple ECId shares many of the features of the low byte of an extended ECId, and is as follows:

7	6	5	4	3	2	1	0
A	ID[3]	ID[2]	ID[1]	ID[0]	FIQ	0	IRQ

Bit(s)	Value	Meaning
A	0	Acorn conformant expansion card
	1	non-conformant expansion card
ID[3:0]	not 0	ID field
	(0	extended ECId used)
FIQ	0	not requesting FIQ
	1	requesting FIQ
IRQ	0	not requesting IRQ
	1	requesting IRQ

Acorn conformance bit

The most significant bit in a simple ECId must be zero for expansion cards that conform to this Acorn specification.

ID field

If you are using a simple ECId, the 4 ID bits may be used for expansion card identification. They must be non-zero, as a value of zero shows that you are instead using an extended ECId.

Interrupt status bits

The interrupt status bits (IRQ and FIQ) are discussed below in a separate section.

Expansion card presence

All expansion cards **must have bit 1 low** in the low byte of the ECId, so that RISC OS can tell if there are any expansion cards present.

Normally bit 1 of the I/O data bus is pulled high by a weak pullup. Therefore if no expansion card is present and RISC OS tries to read the ECId low byte, bit 1 will be set. If a card is present, and the ECId is mapped into memory (which it must be immediately after a reset), the bit will instead be clear.

Extended Expansion Card Identity

If the ID field of the ECId low byte is zero, then the ECId is extended. This means that RISC OS will read the next seven bytes of the ECId. The extended ECId starts at the bottom of the expansion card space, and consists of eight bytes as defined below:

7	6	5	4	3	2	1	0	
C[7]	C[6]	C[5]	C[4]	C[3]	C[2]	C[1]	C[0]	&1C
M[15]	M[14]	M[13]	M[12]	M[11]	M[10]	M[9]	M[8]	&18
M[7]	M[6]	M[5]	M[4]	M[3]	M[2]	M[1]	M[0]	&14
P[15]	P[14]	P[13]	P[12]	P[11]	P[10]	P[9]	P[8]	&10
P[7]	P[6]	P[5]	P[4]	P[3]	P[2]	P[1]	P[0]	&0C
R	R	R	R	R	R	R	R	&08
R	R	R	R	W[1]	W[0]	IS	CD	&04
A	0	0	0	0	FIQ	0	IRQ	&00

Bit(s)	Value	Meaning
C[7:0]		Country (see below)
M[15:0]		Manufacturer (see below)
P[15:0]		Product Type (see below)
R	0	mandatory at present
	1	reserved for future use
W[1:0]	0	8-bit code follows after byte 15 of Id space
	1	16-bit code follows after byte 15 of Id space
	2	32-bit code follows after byte 15 of Id space
	3	reserved
IS	0	no Interrupt Status Pointers follow ECId
	1	Interrupt Status Pointers follow ECId
CD	0	no Chunk Directory follows
	1	Chunk Directory follows Interrupt Status ptrs
A	0	Acorn conformant expansion card
	1	non-conformant expansion card
FIQ	0	not requesting FIQ (or FIQ relocated)
	1	requesting FIQ
IRQ	0	not requesting IRQ (or IRQ relocated)
	1	requesting IRQ

Country code

Every expansion card should have a code for the country of origin. These match those used by the International module, save that the UK has a country code of 0 for expansion cards. If you do not already know the correct country code for your country, you should consult Acorn.

Manufacturer code

Every expansion card should have a code for manufacturer. If you have not already been allocated one, you should consult Acorn.

Product type code

Every expansion card type must have a unique number allocated to it. Consult Acorn if you need to be allocated a new product type code.

Reserved fields

Reserved fields must be set to zero to cater for future expansion.

Width field	For a discussion of the width field, see the earlier section on <i>Expansion Card Identity space</i> .
Generating interrupts	Expansion cards must provide two status bits to show if the card is requesting IRQ or FIQ.
with a simple ECId	If an expansion card only has a simple ECId, then the FIQ and IRQ status bits are bits 2 and 0 respectively in the ECId. If the card does not generate one or both of these interrupts then the relevant bit(s) must be driven low.
with an extended ECId	<p>If an expansion card has an extended ECId, you must set the IS bit of the ECId and provide <i>interrupt status pointers</i> (see below) if either of the following applies:</p> <ul style="list-style-type: none"> • you are also using Chunk Directories (see below) • you want to relocate the interrupt status bits from the low byte of the ECId. <p>If neither of the above apply, then you can omit the Interrupt Status Pointers. The interrupt status bits are located in the low byte of the ECId, and are treated in exactly the same way as for a simple ECId (see above).</p>
Finding out more	<p>To find out more about generating interrupts from expansion cards under RISC OS, you can:</p> <ul style="list-style-type: none"> • see the chapters entitled <i>ARM Hardware</i> and <i>Interrupts and handling them</i> • consult the VL86C010 32-Bit RISC MPU and Peripherals User's Manual, published by Prentice Hall • consult the datasheets for any components you use • contact Customer Support and Services for further hardware-specific details.

Interrupt Status Pointers

An Interrupt Status Pointer has two 4 byte numbers, each consisting of a 3 byte address field and a 1 byte position mask field. These numbers give the locations of the FIQ and IRQ status bits:

IRQ Status Bit address (24 bits)	&40
IRQ Status Bit position mask	&34
FIQ Status Bit address (24 bits)	&30
FIQ Status Bit position mask	&24
	&20

The 24-bit address field must contain signed 2's-complement number giving the offset from &3240000 (the base of the area of memory into which modules are mapped). Hence the cycle speed to access the status register can be included in the offset (encoded by bits 19 and 20). Bits 14 and 15 (that encode the slot number) should be zero. If the status register is in module space then the offset should be negative: eg &DC0000, which is $-\&240000$.

The 8-bit position mask should only have a single bit set, corresponding to the position of the interrupt status bit at the location given by the address field.

Note that these eight bytes are always assumed to be byte-wide. Only the lowest byte in each word should be used.

The addresses may be the same (ie the status bits are in the same byte), so long as the position masks differ. An example of this is if you have had to provide an Interrupt Status Pointer, but do not want to relocate the status bits from the low byte of the ECId; the address fields will both point to the low byte of the ECId, the IRQ mask will be 1, and the FIQ mask will be 4.

If the card does not generate IRQ or FIQ

If the card does not generate one or both of these interrupts then you must set to zero:

- the corresponding address field(s) of the interrupt status pointer
- the corresponding position mask field(s) of the interrupt status pointer
- the corresponding status bit(s) in the low byte of the ECId.

Chunk directory structure

If the CD bit of the extended ECId is set, then:

- the IS bit of the ECId must also be set
- Interrupt Status Pointers must be defined
- a directory of *Chunks* of data and/or code stored in the expansion card's ROM follow the Interrupt Status Pointers.

The lengths and types of these Chunks and the manner in which they are loaded is variable, so after the eight bytes of Interrupt Status Pointers there follow a number of entries in the Chunk Directory. The Chunk Directory entries are eight bytes long and all follow the same format. There may be any number of these entries. This list of entries is terminated by a block of four bytes of zeros.

You should note that, from the start of the Chunk Directory onwards, the width of the expansion card space is as set in the ECId width field. From here on the definition is in terms of bytes:

Start address: 4 bytes (32 bits)	n+8
Size in bytes: 3 bytes (24 bits)	n+4
Operating System identity byte	n+1
	n

The start address is an offset from the base of the expansion card's identity space.

Operating System Identity Byte

The Operating System Identity Byte forms the first byte of the Chunk Directory entry, and determines the type of data which appears in the Chunk to which the Chunk Directory refers. It is defined as follows:

7	6	5	4	3	2	1	0
OS[3]	OS[2]	OS[1]	OS[0]	D[3]	D[2]	D[1]	D[0]

OS[3]	0	reserved
OS[3]	1	mandatory at present
OS[2:0]	0	Acorn Operating System 0: Arthur/RISC OS
	D[3:0]	0 Loader
		1 Relocatable Module
		2 BBC ROM
		3 Sprite
		4 - 15 reserved
	1	reserved
	D[3:0]	0 - 15 reserved
	2	Acorn Operating System 2: Unix
	D[3:0]	0 Loader
		1 - 15 reserved
	3 - 5	reserved
	D[3:0]	0 - 15 reserved
	6	manufacturer defined
	D[3:0]	0 - 15 manufacturer specific
	7	device data
	D[3:0]	0 link
		(for 0, the object pointed to is another directory)
		1 serial number
		2 date of manufacture
		3 modification status
		4 place of manufacture
		5 description
		6 part number
		(for 1 - 6, the data in the pointed-to location contains the ASCII string of the information.)
		7 - 14 reserved
		15 empty chunk

Those Chunks with OS[0:2] = 7, are operating system independent and are always treated as ASCII strings terminated with a zero byte. They are not intended to be read by programs, but rather inspected by users. It is expected that even minimum expansion cards will have an entry for D[3:0] = 5 (description), and it is this string which is printed out by the command *Podules.

Binding a ROM image

For the ROM to be read by the Expansion Card Manager it must conform to the specification, even if only minimally. The simplest way to generate ROM images is to use a BASIC program to combine the various parts together and to compute the header and Chunk Directory structure. Such a program is shown at the end of this chapter. Its output is a file suitable for programming into a PROM or an EPROM.

Code Space

The above forms the basis of storing software and data in expansion cards. However, there is an obvious drawback in that the expansion card space is only 4 kbytes (at word boundaries), and so its usefulness is limited as it stands. To allow expansion cards to accommodate more than this 4 kbytes an extension of the addressing capability is used. This extension is called the Code Space.

The Code Space is an abstracted address space that is accessed in an expansion card independent way via a software interface. It is a large linear address space that is randomly addressable to a byte boundary. This will typically be used for driver code for the expansion card, and will be downloaded into system memory by the operating system before it is used. The manner in which this memory is accessed is variable and so it is accessed via a loader.

Writing a loader

The purpose of the loader is to present to the Expansion Card Manager a simple interface that allows the reading (and writing) of the Code Space on a particular expansion card. The usual case is a ROM paged to appear in 2 kbyte pages at the bottom of the expansion card space, with the page address stored in a latch. This then permits the Expansion Card Manager to load software (Relocatable Modules) or data from an expansion card without having to know how that particular expansion card's hardware is arranged.

The loader is a simple piece of relocatable code with four entry points and clearly defined entry and exit conditions. The format of the loader is optimised for ease of implementation and small code size rather than anything else.

Registers

The register usage is the same for each of the four entry points.

	Input/Output	Comments
R0	Write/Read data	Treated as a byte
R1	Address	Must be preserved
R2-R3		May be used
R4-R9		Must be preserved
R10		May be used
R11	Hardware	Combined hardware address, must be preserved
R12		Private, must be preserved
R13	sp	Stack pointer (FD), must be preserved
R14		Return address; use BICS pc, lr, #V_bit
R15		PC

The exception to this is the CallLoader entry point where R0 - R2 are the user's entry and exit data.

Entry points

All code must be relocatable and position independent. It can be assumed that the code will be run in RAM in SVC mode.

Origin + &00	Read a byte
Origin + &04	Write a byte
Origin + &08	Reset to initial state
Origin + &0C	SWI Podule_CallLoader

Initialisation

The first call made to the loader will be to Read address 0, the start of a Chunk directory for the Code Space.

Errors

Errors are returned in the usual way; V is set and R0 points at a word-aligned word containing the error number, which is followed by an optional error string, which in turn must be followed by a zero byte. ReadByte and WriteByte may be able to return errors like 'Bad address' if the device is not

as big as the address given, or 'Bad write' if using read after write checks on the WriteByte call. If the CallLoader entry is not supported then don't return an error. If Reset fails then return an error.

Since your device drivers may well be short of space, you can return an error with R0=0. The Expansion Card Manager will then supply a default message. Note that this is not encouraged, but is offered as a suggestion of last resort. Errors are returned to the caller by using ORRS pc, lr, #V_bit rather than the usual BICS exit.

Example

Here is an example of a loader (this example, like all others in this chapter, uses the ARM assembler rather than the assembler included with BBC BASIC V – there are subtle syntax differences):

```

00                                LEADR  &FFFFFFD00      ; Data
00 00003000 PageReg *             &3000
00 0000000B PageSize *           11                ; Bits
00 EA00000B Origin B             ReadByte
04 EA000019 B                    WriteByte
08 EA000001 B                    Reset
0C E3DEF201 BICS                 pc, lr, #V_bit
10 FFFFFFFF Page DCD             -1                ; Variable
14 E59FA0E4 Reset LDR            r10, =2_00000011111111111111000000000000
18 E00BA00A AND                  r10, r11, r10      ; Get hardware address from combined on
1C E28AAA03 ADD                  r10, r10, #PageReg
20 E3E02000 MOV                    r2, #-1
24 E50F201C STR                   r2, Page
28 E3A02000 MOV                    r2, #0
2C E4CA2000 STRB                 r2, { r10 }
30 E3DEF201 BICS                 pc, lr, #V_bit
34 E59F40C4 ReadByte LDR         r4, =2_00000011111111111111000000000000
38 E00B4004 AND                  r4, r11, r4        ; Get hardware address from combined on
3C E284AA03 ADD                  r10, r4, #PageReg
40 E3510B3E CMP                    r1, #&F800        ; Last page
44 228F0060 ADRHS                 Error, ErrorATB
48 239EF201 ORRHSS                pc, lr, #V_bit
4C E2812B02 ADD                    r2, r1, #1 :SHL: PageSize
50 E1A025C2 MOV                    r2, r2, ASR #PageSize
54 E51F304C LDR                    r3, Page
58 E1320003 TEQ                    r2, r3
5C 14CA2000 STRNEB                r2, [ r10 ]
60 150F2058 STRNE                 r2, Page
64 E3C12BFE BIC                      r2, r1, #&7F :SHL: PageSize
68 E7D40102 LDRB                   r0, [ r4, r2, ASL #2 ] ; Word addressing
6C E3DEF201 BICS                 pc, lr, #V_bit
70 E28F0000 WriteByte ADR          Error, ErrorNW
74 E39EF201 ORRS                 pc, lr, #V_bit
78 00000580 ErrorNW DCD           ErrorNumber_NotWriteable
A8                                DCB            ErrorString_NotWriteable

```

```

A9 00          DCB    0
AA 00 00      ALIGN
AC 00000584 ErrorATB DCD  ErrorNumber_AddressTooBig
BC           DCB    ErrorString_AddressTooBig
BF 00          DCB    0
CO           END

```

The bit masks are used to separate the fields of a combined hardware address – see the description of `Podule_HardwareAddress` (SWI &40289) for details of these.

Loading the loader

If the Expansion Card Manager is ever asked to 'EnumerateChunk' a Chunk containing a Loader, it will automatically load the Loader. Since RISC OS enumerates all Chunks from all expansion cards at a hard reset this is achieved by default.

If no Loader is loaded then `Podule_EnumerateChunk` will terminate on the zero at the end of the Chunk Directory in the expansion card space. If, however, when the end of the expansion card space Chunk Directory is reached a Loader has been loaded, then a second Chunk Directory, stored in the Code Space, will appear as a continuation of the original Chunk Directory. This is transparent to the user.

This second Chunk Directory is in exactly the same format as the original Chunk Directory. Addresses in the Code Space Chunk Directory refer to addresses in the Code Space. The Chunk Directory starts at address 0 of the Code Space (rather than address 16 as the one in expansion card Space does).

CMOS RAM

Each of the four possible internal expansion card slots has four bytes of CMOS RAM reserved for it. These bytes can be used to store status information, configuration, and so on.

You can find the base address of these four bytes by calling `Podule_HardwareAddress` (SWI &40289).

'Podules'

In the Arthur operating system, expansion cards were known as *Podules*. The word 'Podule' was used in all the names of SWIs and * Commands.

These old names have been retained, so that software written to run under Arthur will still run under RISC OS.

Podule_ReadID (SWI &40280)

SWI calls

	Reads an expansion card's identity byte
On entry	R3 = expansion card slot number
On exit	R0 = expansion card identity byte (ECId)
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 reads into R0 a simple Expansion Card Identity, or the low byte of an extended Expansion Card Identity.
Related SWIs	Podule_ReadHeader (SWI &40281)
Related vectors	None

Podule_ReadHeader (SWI &40281)

Reads an expansion card's header

On entry

R2 = pointer to buffer of 8 or 16 bytes

R3 = expansion card slot number

On exit

—

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 reads an extended Expansion Card Identity into the buffer pointed to by R2. If the IS bit is set (bit 1 of byte 1) then the expansion card also has Interrupt Status Pointers, and these are also read into the buffer.

If you do not know whether the card has Interrupt Status Pointers, you should use a 16 byte buffer.

Related SWIs

Podule_ReadID (SWI &40280)

Related vectors

None

Podule_EnumerateChunks (SWI &40282)

Reads information about a chunk from the chunk directory

On entry

R0 = chunk number (zero to start)
R3 = expansion card slot number

On exit

R0 = next chunk number (zero if final chunk enumerated)
R1 = size (in bytes) if R0 ≠ 0 on exit
R2 = operating system identity byte if R0 ≠ 0 on exit
R4 = pointer to name if the chunk is a relocatable module, else preserved

Interrupts

Interrupt status is unaltered by the SWI, but may be altered by the loader
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads information about a chunk from the chunk directory. It returns its size and operating system identity byte. If the chunk is a module it also returns a pointer to its name; this is held in the Expansion Card Manager's private workspace and will not be valid after you have called the Manager again..

If the chunk is a Loader, then RISC OS also loads it.

To read information on all chunks you should set R0 to 0 and R3 to the expansion card's slot number. You should then repeatedly call this SWI until R0 is set to 0 on exit. RISC OS automatically does this on a reset for all cards; if there is a Loader it will be transparently loaded, and any chunks in the code space will also be enumerated.

Related SWIs

Podule_ReadChunk (SWI &40283)

Related vectors

None

Podule_ReadChunk (SWI &40283)

Reads a chunk from an expansion card

On entry

R0 = chunk number
R2 = pointer to buffer (assumed large enough)
R3 = expansion card slot number

On exit

—

Interrupts

Interrupt status is unaltered by the SWI, but may be altered by the loader
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads the specified chunk from an expansion card. The buffer must be large enough to contain the chunk; you can use Podule_EnumerateChunks (SWI &40282) to find the size of the chunk.

Related SWIs

Podule_EnumerateChunks (SWI &40282)

Related vectors

None

Podule_ReadBytes (SWI &40284)

Reads bytes from within an expansion card's code space

On entry

R0 = offset from start of code space
R1 = number of bytes to read
R2 = pointer to buffer
R3 = expansion card slot number

On exit

—

Interrupts

Interrupt status is unaltered by the SWI, but may be altered by the loader
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call reads bytes from within an expansion card's code space. It does so using repeated calls to offset 0 (read a byte) of its Loader.

RISC OS must already have loaded the Loader; note that this is done automatically on a reset.

Related SWIs

Podule_WriteBytes (SWI &40285)

Related vectors

None

Podule_WriteBytes (SWI &40285)

Writes bytes to within an expansion card's code space

On entry

R0 = offset from start of code space

R1 = number of bytes to write

R2 = pointer to buffer

R3 = expansion card slot number

On exit

—

Interrupts

Interrupt status is unaltered by the SWI, but may be altered by the loader
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This call writes bytes to within an expansion card's code space. It does so using repeated calls to offset 4 (write a byte) of its Loader.

RISC OS must already have loaded the Loader; note that this is done automatically on a reset.

Related SWIs

Podule_ReadBytes (SWI &40284)

Related vectors

None

Podule_CallLoader (SWI &40286)

	<p>Calls an expansion card's Loader</p>
On entry	<p>R0 - R2 = user data R3 = expansion card slot number</p>
On exit	<p>R0 - R2 = user data</p>
Interrupts	<p>Interrupt status is unaltered by the SWI, but may be altered by the loader Fast interrupts are enabled</p>
Processor mode	<p>Processor is in SVC mode</p>
Re-entrancy	<p>Depends on loader</p>
Use	<p>This call enters an expansion card's Loader at offset 12. Registers R0 - R2 can be used to pass data.</p> <p>The action the Loader takes will vary from card to card, and you should consult your card's documentation for further details.</p> <p>If you are developing your own card, you can use this SWI as an entry point to add extra features to your Loader. You may use R0 - R2 to pass any data you like. For example, R0 could be used as a reason code, and R1 and R2 to pass data.</p>
Related SWIs	<p>None</p>
Related vectors	<p>None</p>

Podule_RawRead (SWI &40287)

Reads bytes directly within an expansion card's address space

On entry

R0 = offset from base of a podule's address space (0...&3FFF)

R1 = number of bytes to read

R2 = pointer to buffer

R3 = expansion card slot number

On exit

—

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 reads bytes directly within an expansion card's address space. It is typically used to read from the registers of hardware devices on an expansion card.

Podule_ReadBytes (SWI &40284) should be used to read within the card's code space.

Related SWIs

Podule_RawWrite (SWI &40288)

Related vectors

None

Podule_RawWrite (SWI &40288)

Writes bytes directly within an expansion card's address space

On entry

R0 = offset from base of a podule's address space (0...&3FFF)

R1 = number of bytes to write

R2 = pointer to buffer

R3 = expansion card slot number

On exit

—

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 writes bytes directly within an expansion card's address space. It is typically used to write to the registers of hardware devices on an expansion card.

Podule_WriteBytes (SWI &40285) should be used to write within the card's code space.

Related SWIs

Podule_RawRead (SWI &40287)

Related vectors

None

Podule_HardwareAddress (SWI &40289)

Returns an expansion card's base address, and the address of its CMOS RAM

On entry

R3 = expansion card slot number or expansion card base address

On exit

R3 = combined hardware address

Interrupts

Interrupt status is unaltered
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call returns an expansion card's combined hardware address:

Bits	Meaning
0 - 11	base address of CMOS RAM (4 bytes)
12 - 25	bits 12 - 25 of base address of expansion card
26 - 31	reserved

You can use a mask to extract the relevant parts of the returned value. The CMOS address in the low 12 bits is suitable for passing directly to OS_Byte 161 and 162.

Related SWIs

OS_Byte 161 and 162 (SWI &06)

Related vectors

None

*PoduleLoad

* Commands

Copies a file into the RAM area of an expansion card

Syntax

```
*PoduleLoad <expansion card number> <filename> [<offset>]
```

Parameters

<expansion card number> the expansion card's number, as given by *Podules
<filename> a valid pathname, specifying a file
<offset> offset into space accessed by Loader

Use

*PoduleLoad copies the contents of a file into the RAM area of an installed expansion card, starting at the specified offset. If no offset is given, then a default value of 0 is used.

Example

```
*PoduleLoad 1 $.Midi.Data 256
```

Related commands

*Podules, *PoduleSave

Related SWIs

Podule_WriteBytes (SWI &40285)

Related vectors

None

*Podules

	Tells you which expansion cards are installed
Syntax	*Podules
Parameters	None
Use	<p>This command tells you which expansion cards are installed using the description that each one holds internally. Some expansion cards, such as one that is still being designed, will not have a description; in this case, an identification number is printed.</p> <p>This command still refers to expansion cards as podules, to maintain compatibility with the Arthur operating system.</p>
Example	<pre>*Podules Podule 0: Midi and BBC I/O podule Podule 1: Simple podule &8 Podule 2: No installed podule Podule 3: No installed podule</pre>
Related commands	None
Related SWIs	Podule_EnumerateChunks (SWI &40282)
Related vectors	None

*PoduleSave

Copies the contents of an expansion card's ROM into a file

Syntax

```
*PoduleSave <expansion card number> <filename> <size>  
[<offset>]
```

Parameters

<expansion card number> the expansion card's number, as given by
*Podules
<filename> a valid pathname, specifying a file
<size> in bytes
<offset> offset into space accessed by Loader

Use

*PoduleSave copies the given number of bytes of an installed expansion card's ROM into a file. If no offset is given, then a default value of 0 is used.

Example

```
*PoduleSave 1 $.Midi.Data 512 256
```

Related commands

*Podules, *PoduleLoad

Related SWIs

Podule_ReadBytes (SWI &40284)

Related vectors

None

Example program

This program is an example of how to combine the various parts of an expansion card ROM. It also computes the header and Chunk Directory structure. The file it outputs is suitable for programming into a PROM or EPROM:

```
10 REM > 4.arm.MidiAndI/O.MidiJoiner
20 REM Author : RISC OS
30 REM Last edit : 06-Jan-87
40 PRINT"Joiner for expansion card ROMs""Version 1.05."
50 PRINT"For Midi board.": DIM Buffer% 300, Block% 20
70 INPUT"Enter name of output file : "OutName$
75 H%=OPENOUT(OutName$)
80 IF H%=0 THEN PRINT"Could not create '"+OutName$;""":END
90 ONERRORONERROROFF:CLOSE#H%:REPORT:PRINT" at line ";ERL:END
100 Device%=0:L%=TRUE:REPEAT
120 Max%=&800:REM Max% is the size of the normal area
130 Low%=&100:REM Low% is the size of the pseudo directory
140 Base%=0:REM The offset for file address calculations
150 Rom%=&4000:REM Rom% is the size of BBC ROMs
170 PROCByte(0):PROCHalf(3):PROCHalf(19):PROCHalf(0):PROCByte(0)
180 PROCByte(0):PROC3Byte(0):PROCByte(0):PROC3Byte(0)
190 IF PTR#H% <> 16 STOP
200 Bot%=PTR#H%:REM Bot% is where the directory grows from
210 Top%=Max%:REM Top% is where normal files descend from
230 INPUT"Enter filename of loader : "Loader$
240 IF Loader$ <> "" THEN K%=FNAddFile( &80, Loader$ )
250 IF K% ELSE PRINT"No room for loader.":
    PTR#H%=Bot%:PROCByte(0):CLOSE#H%:END
270 INPUTLINE"Enter product description : "Dat$
280 IF Dat$ <> "" THEN PROCAddString( &F5, Dat$ )
300 PRINT:REPEAT
310 INPUT"Enter name of file to add : "File$
320 IF File$ <> "" THEN T%=FNType( File$ ) ELSE T%=0
330 IF T%=0 ELSE K%=FNAddFile( T%, File$ )
340 IF K% ELSE PRINT"No more room."
350 UNTIL (File$ = "") OR (K%=FALSE)
360 IF K% ELSE PTR#H%=Bot%:PROCByte(0):CLOSE#H%:END
370 IF L% PROCChange
390 INPUTLINE"Enter serial number : "Dat$
400 IF Dat$ <> "" THEN PROCAddString( &F1, Dat$ )
410 INPUTLINE"Enter modification status : "Dat$
420 IF Dat$ <> "" THEN PROCAddString( &F3, Dat$ )
430 INPUTLINE"Enter place of manufacture : "Dat$
440 IF Dat$ <> "" THEN PROCAddString( &F4, Dat$ )
450 INPUTLINE"Enter part number : "Dat$
460 IF Dat$ <> "" THEN PROCAddString( &F6, Dat$ )
480 Date$=TIMES
490 Date$=MIDS(Date$,5,2)+"-"+MIDS(Date$,8,3)+"-"+MIDS(Date$,14,2)
500 PROCAddString( &F2, Date$ )
530 REM PROCHeader( &F0, 2%+W%*Rom%-Base%, 0 ):REM Link
550 PTR#H%=Bot%:PROCByte(0)
570 CLOSE#H%: END
```

```

590 DEF PROCByte(D%):BPUT#H%,D%:ENDPROC
610 DEF PROCHalf(D%):BPUT#H%,D%:BPUT#H%,D%DIV256:ENDPROC
630 DEF PROC3Byte(D%)
640 BPUT#H%,D%:BPUT#H%,D%DIV256:BPUT#H%,D%DIV65535:ENDPROC
660 DEF PROCWord(D%)
670 BPUT#H%,D%:BPUT#H%,D%DIV256:BPUT#H%,D%DIV65535
680 BPUT#H%,D%DIV16777216:ENDPROC
700 DEF PROCAddString( T%, SS )
710 SS=SS+CHR$0
720 IF L% THEN PROCAddNormalString ELSE PROCAddPseudoString
730 ENDPROC
750 DEF PROCAddNormalString
760 IF Top%-Bot% < 10+LEN(SS) THEN STOP
770 PROCHeader( T%, Top%-LEN(SS)-Base%, LEN(SS) )
780 Top%=Top%-LEN(SS):PTR#H%=Top%:FOR I%=1 TO LEN(SS)
790 BPUT#H%,ASC(MIDS(SS,I%,1)):NEXTI%:ENDPROC
810 DEF PROCAddPseudoString
820 IF Max%+Low%-Bot% < 9 THEN STOP
830 PROCHeader( T%, Top%-Base%, LEN(SS) )
840 PTR#H%=Top%:FOR I%=1 TO LEN(SS)
850 BPUT#H%,ASC(MIDS(SS,I%,1)):NEXTI%
860 Top%=Top%+LEN(SS):ENDPROC
880 DEF PROCHeader( Type%, Address%, Size% )
890 PTR#H%=Bot%
900 PROCByte( Type% )
910 PROC3Byte( Size% )
920 PROCWord( Address% )
930 Bot%=Bot%+8:ENDPROC
950 DEF FNAddFile( T%, N$ )
960 F%=OPENIN( N$ )
970 IF F%=0 THEN PRINT"File '"+N$;" not found.":=FALSE
980 S%=EXT#F%
990 IF L% THEN =FNAddNormalFile ELSE =FNAddPseudoFile
1010 DEF FNAddNormalFile
1020 E%=S%+9-(Top%-Bot%)
1030 IF E%>0 THEN PRINT"Oversize by ";E%;" bytes.":
PROCChange:=FNAddPseudoFile
1040 PROCHeader( T%, Top%-S%-Base%, S% )
1050 Top%=Top%-S%:PTR#H%=Top%:FOR I%=1 TO S%
1060 BPUT#H%,BGET#F%:NEXTI%:CLOSE#F%:=TRUE
1080 DEF FNAddPseudoFile
1090 IF Max%+Low%-Bot% < 9 THEN =FALSE
1100 PROCHeader( T%, Top%-Base%, S% )
1110 PTR#H%=Top%
1120 FOR I%=1 TO S%:BPUT#H%,BGET#F%:NEXTI%
1130 Top%=Top%+S%:CLOSE#F%:=TRUE
1150 DEF PROCChange
1160 PRINT"Changing up. Wasting ";Top%-Bot%;" bytes."
1170 PTR#H%=Bot%:PROCByte(0):REM Terminate bottom directory
1180 Bot%=Max%:Top%=Max%+Low%:Base%=Max%:L%=FALSE
1190 REM In the pseudo area files grow upward from Top%
1200 ENDPROC
1220 DEF FNTYPE( N$ )

```

```
1230 $Buffer% = NS: X% = Block%: Y% = X% / 256: A% = 5: X% ! 0 = Buffer%
1240 B% = USR&FFDD: IF (B% AND 255) <> 1 THEN PRINT "Not a file": = 0
1250 V% = (Block% ! 3) AND &FFFFFF
1260 IF V% = &FFFFFFA THEN = &B1
1270 IF ((Block% ! 2 AND &FFFF) = &B000) AND ((Block% ! 6 AND &FFFF) = &B000) THEN = &B2
1280 IF V% = &FFFFFF9 THEN = &B3
1290 = 0
```

International module

Introduction

The International module allows the user to tailor the machine for use in different countries by setting:

- the keyboard – the mapping of keys to character codes
- the alphabet – the mapping from character codes to characters
- the country – both of the above mappings.

This module, in conjunction with the RISC OS kernel, controls the selection of these mappings, but it allows the actual mappings to be implemented in other modules, via the service mechanism. Thus, you could write your own international handlers.

Each country is represented by a name and number. The keyboard shares this list, and is normally on the same setting. However, there are cases for the country and the keyboard to be different. For example, the Greek keyboard would not allow you to type *Commands, because only Greek characters could be entered. In this case, the country could remain Greek, while the keyboard setting is changed temporarily for *Commands.

Each alphabet is also represented by a name and number. A country can only have one alphabet associated with it, but an alphabet can be used by many countries. For example, the Latin1 alphabet contains a general enough set of characters to be used by most Western European countries.

Overview and Technical Details

Names and numbers

Country numbers range from 0 to 99, and alphabet numbers are from 100 to 126. Here are lists of the currently available countries and alphabets.

Countries and keyboards

Here is a list of the currently-defined country and keyboard codes (provided by the international module), and the alphabets they use:

Code	Country and Keyboard	Alphabet
0	Default	Selects the configured country. If the configured country is 'Default', the keyboard ID byte is read from the keyboard
1	UK	Latin1
2	Master	BFont
3	Compact	BFont
4	Italy	Latin1
5	Spain	Latin1
6	France	Latin1
7	Germany	Latin1
8	Portugal	Latin1
9	Esperanto	Latin3
10	Greece	Greek
11	Sweden	Latin1
12	Finland	Latin1
13	(not used)	
14	Denmark	Latin1
15	Norway	Latin1
16	Iceland	Latin1
17	Canada1	Latin1
18	Canada2	Latin1
19	Canada	Latin1
20	Turkey	Latin3
21	Arabic	Special - ISO 8859/6
22	Ireland	Latin1
23	Hong Kong	<i>Not defined at time of going to press</i>

80	ISO1	Latin1
81	ISO2	Latin2
82	ISO3	Latin3
83	ISO4	Latin4

Alphabets

Here is a list of the alphabet codes currently defined, provided by the international module:

Code	Alphabet
100	BFont
101	Latin1
102	Latin2
103	Latin3
104	Latin4
107	Greek

Alphabet

OS_Byte 71 (SWI &06) reads or sets the alphabet by number. *Alphabet can also set the alphabet by name. *Alphabets lists all the available alphabets on the system. Remember that you should normally only need to change the country setting as this will also change the alphabet.

Use OS_ServiceCall &43,1 (SWI &30) to convert between alphabet name and number forms and OS_ServiceCall &43,3 to convert from alphabet number to name forms.

OS_ServiceCall &43,5 causes a module which recognises the alphabet number to define the characters in an alphabet in the range specified, by issuing VDU 23 commands itself. The call is issued by the OS when OS_Byte 71 is called to set the alphabet and also by OS_Byte 20 and 25.

Keyboard

OS_Byte 71 can also be used to read or set the keyboard number. *Keyboard can set it as well. Remember that you should normally only need to change the country setting as this will also change the keyboard.

When the keyboard setting is changed, by either of the above ways, an OS_ServiceCall &43,6 will be generated automatically. This is a broadcast to all keyboard handler modules that the keyboard selection has changed.

Country

Setting the country will set values for the alphabet and the keyboard. You should not usually have to override these settings. The country number can be read or set with OS_Byte 70. OS_Byte 240 can also read it. *Country can set the country by name. *Countries will list all the available country names. *Configure Country will set the default country by name and store it in CMOS RAM.

Use OS_ServiceCall &43,0 to convert between country name and number forms and OS_ServiceCall &43,2 to convert from country number to name forms.

To get the default alphabet for a country, OS_ServiceCall &43,4 can be called. Remember that the default keyboard number is the same as the country number.

Service calls

RISC OS provides service calls for the use of any module that adds to the set of international character sets and countries. These are described in the chapter entitled *Modules*.

SWI Calls

OS_Byte 70 (SWI &06)

	Read/write country number
On entry	R0 = 70 (&46) (reason code) R1 = 127 to read or country number to write
On exit	R0 is preserved R1 = country number read or before being overwritten, or 0 if invalid country number passed R2 is 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 returns or sets the country number used by the international module.
Related SWIs	OS_Byte 240 (SWI &06)
Related vectors	ByteV

OS_Byte 71 (SWI &06)

Read/write alphabet or keyboard

On entry

R0 = 71 (&47) (reason code)
R1 = 0–126 for setting the alphabet number
127 for reading the current alphabet number
128–254 for setting the keyboard number (R1–128)
255 for reading the current keyboard number

On exit

R0 is preserved
R1 = alphabet or keyboard number read or before being overwritten,
or 0 if invalid value passed
R2 is 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 returns or sets the alphabet or keyboard number used by the international module. Their settings can be read without altering them, or you can set a new value for either. This SWI will return a zero if the value passed to set the new value is not one of the known alphabets or keyboards.

Note that the keyboard setting is offset by 128. eg. to set keyboard 3, you must pass 131 in R1.

Related SWIs

OS_Byte 70 (SWI &06)

Related vectors

ByteV

OS_Byte 240 (SWI &06)

	Read country number
On entry	R0 = 240 (&F0) (reason code) R1 = 0 R2 = 255
On exit	R0 is preserved R1 = country number R2 = user flag (see OS_Byte 241)
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 country number used by the international module.
Related SWIs	OS_Byte 70 (SWI &06)
Related vectors	ByteV

*Commands

*Alphabet

Selects an alphabet

Syntax

```
*Alphabet [<country name> | <alphabet name>]
```

Parameters

<country name> See the *Countries command for a list of countries available

<alphabet name> See the *Alphabets command for a list of alphabets available

Use

*Alphabet sets the alphabetical set of characters according to the country name or alphabet name.

The *Alphabet command with no parameter displays the currently selected alphabet.

Example

```
*Alphabet Latin3
```

Related commands

*Alphabets

Related SWIs

OS_Byte 71 (SWI &06)

Related vectors

None

*Alphabets

List all the alphabets installed

Syntax

*Alphabets

Use

*Alphabets lists all the alphabets currently supported by your Acorn computer. Use the *Alphabet command to change the alphabetical set of characters.

Example

```
*Alphabets
Alphabets:
BFont  Latin1 Latin2 Latin3 Latin4 Greek
```

Related commands

*Alphabet

Related SWIs

OS_Byte 71 (SWI &06)

Related vectors

None

*Configure Country

Sets the default alphabet and keyboard

Syntax

```
*Configure Country <country name>
```

Parameters

<country name> Valid countries are currently Canada, Canada1, Canada2, Compact, Default, Denmark, Esperanto, Finland, France, Germany, Greece, Iceland, ISO1, Italy, Master, Norway, Portugal, Spain, Sweden, and UK. A list of parameters can be obtained with the *Countries command.

Use

*Configure Country sets the appropriate default alphabet and keyboard layout. For some countries you also need to load a relocatable module to define the keyboard layout. If the configured country is Default, then the keyboard ID byte (read from the keyboard) is used as the country number, providing it is in the range 1–31. Current UK keyboards return keyboard ID 1, which corresponds to country UK.

Example

```
*Configure Country Italy
```

Related commands

```
*Country, *Countries
```

Related SWIs

```
OS_Bytes 70 and 240 (SWI &06)
```

Related vectors

```
None
```

*Country

Sets the appropriate alphabet and keyboard layout for a given country

Syntax

*Country [<country name>]

Parameters

<country name> Use the *Countries command for a list of countries available.

Use

*Country sets both the appropriate alphabet and keyboard layout for a particular country; for example, *Country UK selects the Latin1 alphabet and the UK keyboard layout. *Alphabet and *Keyboard can, however, be used to set the alphabet and keyboard layout independently, leaving the setting for country unchanged.

The *Country command without a parameter displays the currently selected country.

Example

```
*Country Italy
```

Related commands

*Configure Country, *Countries, *Alphabet, *Alphabets, *Keyboard

Related SWIs

OS_Bytes 70 and 240 (SWI &06)

Related vectors

None

*Countries

Lists the available countries

Syntax

*Countries

Use

*Countries lists the countries that are available on the modules currently in the system..

Example

*Countries

Countries:

Default	UK	Master	Compact	Italy	Spain	France
Germany	Portugal		Esperanto		Greece	Sweden
Norway	Iceland	Canadal	Canada2	Canada	ISO1	

Related commands

*Configure Country, *Country, *Alphabet, *Alphabets, *Keyboard

Related SWIs

OS_Bytes 70 and 240 (SWI &06)

Related vectors

None

*Keyboard

	Selects the keyboard driver for a given country
Syntax	*Keyboard [<country name>]
Parameters	<country name>the name of an available country (see *Countries)
Use	Selects a keyboard driver for a particular country. *Keyboard without a parameter displays the currently set keyboard.
Example	*Keyboard Denmark
Related commands	*Country
Related SWIs	OS_Byte 71 (SWI &06)
Related vectors	None

Vertical line

Debugger

Introduction

The debugger is a module that allows program to be stopped at set places called breakpoints. Whenever the instruction that a breakpoint is set on is reached, a command line will be entered. From here, you can type debug commands and resume the program when you want.

Other commands may be called at any time to examine or change the values contained at particular addresses in memory and to list the contents of the registers. You can display memory as words or bytes.

There is also a facility to disassemble instructions. This means converting the instruction, stored as a word into a string representation of its meaning. This allows you to examine the code anywhere in readable memory.

Technical Details

The debugger provides one SWI, `Debugger_Disassemble` (SWI &40380), which will disassemble one instruction. There are also the following *Commands:

Command	Description
*BreakClr	Remove breakpoint
*BreakList	List currently set breakpoints
*BreakSet	Set a breakpoint at a given address
*Continue	Start execution from a breakpoint saved state
*Debug	Enter the debugger
*InitStore	Fill memory with given data
*Memory	Display memory between two addresses/register
*MemoryA	Display and alter memory
*MemoryI	Disassemble ARM instructions
*ShowRegs	Display registers caught by traps

When an address is required, it should be given in hex, without a preceding &. That is, unlike most of the rest of the system, the debugger uses hex as a default base rather than decimal.

*Quit should be used to return from the debugger to the previous environment after a breakpoint. This is described in the chapter entitled *Program Environment*.

Note that the breakpoints discussed here are separate from those caused by `OS_BreakPt`. See the chapter entitled *Program Environment* for details of this SWI.

When a breakpoint is set, the previous contents of the breakpoint address are replaced with a branch into the debugger code. This means that breakpoints may only be set in RAM. If you try to set a breakpoint in ROM, the error `Bad breakpoint address` will be given.

When a breakpoint instruction is reached, the debugger is entered, with the prompt `Debug*`, from which you can type any *Command. An automatic register dump is also displayed.

Debugger_Disassemble (SWI &40380)

SWI Calls

Disassemble an instruction

On entry

R0 = instruction to disassemble
R1 = address to assume the word came from

On exit

R0 = preserved
R1 = address of buffer containing null-terminated text
R2 = length of disassembled line

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor Mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

R0 contains the 32-bit instruction to disassemble. R1 contains the address to assume the word came from, which is needed for instructions such as B, BL, LDR Rn, [PC...], and so on. On exit, R1 points to a buffer which contains a zero terminated string. This string consists of the instruction mnemonic, and any operands, in the format used by the *MemoryI instruction. The length in R2 includes the zero-byte.

Related SWIs

None

Related vectors

None

*Commands

*BreakClr

Remove a breakpoint

Syntax

```
*BreakClr [<addr>|<reg>]
```

Parameters

<addr> address to clear breakpoint in hex

<reg> register value to clear breakpoint

Allowed register names are r0 - r15, sp (equivalent to r13), lr (r14 without the psr bits) and pc (r15 without the psr bits). These are taken from the current ExceptionDumpArea.

Use

*BreakClr removes the breakpoint at the specified address/register value, putting the original contents back into that location. You can unset the last hit breakpoint with the command `*BreakClr pc`

If no parameter is given then you can remove all breakpoints – you will be prompted:

```
Clear all breakpoints [Y/<anything>]?
```

Example

```
*BreakClr 7FF6B
```

Related commands

*BreakSet, *BreakList

Related SWIs

None

Related vectors

None

*BreakList

List all breakpoints

Syntax

*BreakList

Use

*BreakList lists all the breakpoints that are currently set with *BreakSet.

Example

*BreakList

Address	Old Data
00008704	EF00141C

Related commands

*BreakSet

Related SWIs

None

Related vectors

None

*BreakSet

Set a breakpoint

Syntax	<code>*BreakSet <addr> <reg></code>
Parameters	<p><code><addr></code> address to set breakpoint in hex</p> <p><code><reg></code> register value to set breakpoint</p> <p>Allowed register names are r0 - r15, sp (equivalent to r13), lr (r14 without the psr bits) and pc (r15 without the psr bits). These are taken from the current ExceptionDumpArea.</p>
Use	<p>*BreakSet sets a breakpoint at the address or register value given, so that when the code is executed and the instruction at that address is reached, execution will be halted.</p> <p>When a breakpoint is set, the previous contents of the breakpoint address are replaced with a branch into the debugger code. This means that breakpoints may only be set in RAM. If you try to set a breakpoint in ROM, the error <code>Bad breakpoint address</code> will be given.</p>
Example	<code>*BreakSet 1665D</code>
Related commands	<code>*BreakClr</code> , <code>*BreakList</code>
Related SWIs	None
Related vectors	None

*Continue

Resume execution after a breakpoint

Syntax

*Continue

Use

*Continue starts execution from the breakpoint saved state. If there is a breakpoint at the continuation position, then this prompt is given:

```
Continue from breakpoint set at &00008704  
Execute out of line? [Y/<anything>]?
```

Reply 'Y' if it is permissible to execute the instruction at a different address (ie it does not refer to the PC). If the instruction that was replaced by the breakpoint contains a PC-relative reference (eg `LDR R0,label` or an `ADR` directive), then you should reset the break point before continuing. This causes the instruction to be executed in-line, otherwise the wrong address is referenced.

Related commands

None

Related SWIs

None

Related vectors

None

*Debug

Enter the debugger

*Debug

This command enters the debugger. A Debug* prompt is given.

Use Escape to return to the caller, or *Quit to exit to the caller's parent.

*Quit is documented in the chapter entitled *Program Environment*.

*Quit

None

None

Syntax

Use

Related commands

Related SWIs

Related vectors

*InitStore

Fill user memory with a value

Syntax

```
*InitStore [<value>|<reg>]
```

Parameters

<value> word to fill user memory with

<reg> register value to fill memory with

Allowed register names are r0 - r15, sp (equivalent to r13), lr (r14 without the psr bits) and pc (r15 without the psr bits). These are taken from the current ExceptionDumpArea.

Use

*InitStore fills user memory with the specified data or the value &E1000090 (which is an illegal instruction) if no parameter is given. If you give this command from within an application (eg BASIC), the machine will crash, and will have to be reset.

Example

```
*InitStore &381E6677
```

Related commands

None

Related SWIs

None

Related vectors

None

*Memory

Display values in memory

Syntax

```
*Memory [B] <addr1>|<reg1> or  
*Memory [B] <addr1>|<reg1> {+|-}<addr2>|<reg2> or  
*Memory [B] <addr1>|<reg1> {+|-}<addr2>|<reg2> +<addr3>|<reg3>
```

Parameters

B optionally display as bytes
<addr1>|<reg1> address or register value for start of display
<addr2>|<reg2> an address or register value
<addr3>|<reg3> an address or register value

Allowed register names are r0 - r15, sp (equivalent to r13), lr (r14 without the psr bits) and pc (r15 without the psr bits). These are taken from the current ExceptionDumpArea.

Use

*Memory displays the values in the memory in words from the address given either explicitly or contained in a register.

If only one address is given 256 bytes are displayed.

If two addresses are given addr2 specifies the end of the range to be displayed (as an offset from addr1).

If three addresses are given, addr2 specifies an offset for the start from addr1, and addr3 specifies the end of the range to be displayed (as an offset from the combined address given by addr1 and addr2).

If the optional B is given after the command but before the start address, the display is byte-oriented, with 16 bytes per line. If it is omitted, the display is word-oriented, with four words per line.

Example

```
*Memory 1000 -200 +500 Display memory from &800 to &1300
```

Related commands

```
*MemoryA, *MemoryI
```

Related SWIs

None

Related vectors

None

*MemoryA

Display and alter memory

Syntax

```
*MemoryA [B] <addr/reg1> [<data>|<reg2>]
```

Parameters

B	optionally display as bytes
<addr1/reg1>	an address or register value
<data>	a value to write into the specified location
<reg2>	register containing value to write

Use

*MemoryA displays and alters the memory in bytes, if the optional B is given, or in words otherwise. It starts at the address given absolutely or within a register. If no further parameters are given, interactive mode is entered where the following may be typed:

Return	to go to 'next' location
-	to step backwards in memory
+	to step forwards in memory
<hex digits>	to alter a location and proceed
.	to exit.

At each line, something similar to the following is printed:

```
+ 000087A0 : cccc : xxxxxxxx : opcode
Enter new value :
```

where the '+' is the direction in which Return steps (it is '-' for backwards). Next is the address of the word/byte being altered, then the four characters in that word, then the current hexadecimal value of the word, and finally the instruction at that address.

In byte mode, it looks like this:

```
+ 000087A1 : c : xx
```

Alternatively you can give the new data value on the line after the address.

Example

```
*MemoryA 87A0 123456578
```

Related commands

```
*Memory, *MemoryI
```

Related SWIs

None

Related vectors

None

*MemoryI

Disassemble memory

Syntax

```
*MemoryI <addr1>|<reg1> or
*MemoryI <addr1>|<reg1> [+|-<addr2>|<reg2> or
*MemoryI <addr1>|<reg1> +|-<addr2>|<reg2> +<addr3>|<reg3>
```

Parameters

B optionally display as bytes
<addr1>|<reg1> address or register value for start of display
<addr2>|<reg2> an address or register value
<addr3>|<reg3> an address or register value

Allowed register names are r0 - r15, sp (equivalent to r13), lr (r14 without the psr bits) and pc (r15 without the psr bits). These are taken from the current ExceptionDumpArea.

Use

*MemoryI disassembles memory from the address given either explicitly or contained in a register.

If only one address is given 25 instructions are disassembled.

If two addresses are given addr2 specifies the end of the range to be disassembled (as an offset from addr1).

If three addresses are given, addr2 specifies an offset for the start from addr1, and addr3 specifies the end of the range to be disassembled (as an offset from the combined address given by addr1 and addr2).

These options are particularly useful for disassembling modules which contain offsets, not addresses.

Example

```
*modules Find address of module
...
32 01828F34 01829D04 ColourTrans
...

*memoryi 01828f34+28 Disassemble header
01828F34 : .... : 00000000 : ANDEQ R0,R0,R0
01828F38 : E... : 000001C8 : ANDEQ R0,R0,R8,ASR #3
01828F3C : ^... : 000001B0 : MULEQ R0,R0,R1
01828F40 : ^... : 00000260 : ANDEQ R0,R0,R0,ROR #4
01828F44 : (... : 00000028 : ANDEQ R0,R0,R8,LSR #32
```

```

01828F48 : 4... : 00000034 : ANDEQ  R0,R0,R4,LSR R0
01828F4C : .... : 00000000 : ANDEQ  R0,R0,R0
01828F50 : @... : 00040740 : ANDEQ  R0,R4,R0,ASR #14 ← Offset of SWI handler is
01828F54 : _... : 00000298 : MULEQ  R0,R8,R2 ← &0298
01828F58 : W... : 00000057 : ANDEQ  R0,R0,R7,ASR R0

*memory! 01828f34+298+40                                     Disassemble SWI handler
018291CC : . ?a : E33FF003 : TEQP   PC,#3
018291D0 : .C-é : E92D4300 : STMDB  R13!,{R8,R9,R14}
018291D4 : ./ á : E1A0800B : MOV    R8,R11
018291D8 : " _ a : E3A09022 : MCV    R9,#522 ; =""
018291DC : 4..Y : EF020034 : SWI    XOS_CallAVector
018291E0 : .C è : E8BD4300 : LDMIA  R13!,{R8,R9,R14}
018291E4 : .â_c : 638EE201 : ORRVS  R14,R14,#61000000
018291E8 : . "á : E1B0F00E : MOVS   PC,R14
018291EC : .@ è : E8BD4000 : LDMIA  R13!,{R14}
018291F0 : ..Xa : E3580011 : CMP    R8,#611 ; =17
018291F4 : .h_ : B08FF108 : ADDLT  PC,PC,R8,LSL #2
018291F8 : ...è : EAC00010 : B      01829240
018291FC : a..è : EAC00091 : B      01829448
01829200 : !..è : EAC000A6 : B      018294A0
01829204 : W..è : EA000057 : B      01829368
01829208 : 5..è : EA000035 : B      018292E4

```

Related commands

*MemoryA, *MemoryI

Related SWIs

None

Related vectors

None

*ShowRegs

Display register contents

Syntax

```
*ShowRegs
```

Use

*ShowRegs displays the registers caught on one of the five following traps:

- unknown instruction
- address exception
- data abort
- abort on instruction fetch
- breakpoint.

It also prints the address in memory where the registers are stored, so you can alter them (for example after a breakpoint) by using *MemoryA on these locations, before using *Continue.

Example

```
*ShowRegs
Register dump (stored at 401804D2C) 1s:
R0 = 0026D2CF R1 = 002483C1 R2 = 00000000 R3 = 00000000
R4 = 00000000 R5 = 52491ACE R6 = 42538FFD R7 = 263598DE
R8 = B278A456 R9 = C2671D37 R10 = A72B34DC R11 = 82637D2F
R12 = 00004000 R13 = 2538DAFD R14 = 24368000 R15 = 7629D100
Mode USR flags set : nzcvi f
```

Related commands

None

Related SWIs

None

Related vectors

None

1694

Floating point emulator

Introduction

Certain Acorn RISC machines support a general co-processor interface. The optional hardware floating point co-processor (contact your supplier for availability) performs floating point calculations to IEEE standard 754.

RISC OS also contains a floating point software emulator module which provides floating point support. The instructions it provides may be incorporated into any assembler text, provided they are called from user mode. However, these instructions are not supported by the BASIC interpreter.

Because this module doesn't present any SWIs or other usual interface to programs (apart from a SWI to return the version number), it is structured differently from the others. First, there is a discussion of the programmer's model of the IEEE 754 floating point system. This is followed by the floating point instruction set. Finally the SWI is detailed.

Note that the floating point co-processor only directly supports a sub-set of the instruction repertoire, the remainder still being emulated in software, which also range-reduces trigonometric function arguments before they are executed by the hardware.

Generally, programs do not need to know whether a co-processor is fitted; the only effective difference is in the speed of execution. Note that there may be slight variations in accuracy between hardware and software – refer to the instructions supplied with the co-processor for details of these variations.

Programmer's model

The ARM IEEE floating point system has eight 'high precision' floating point registers, F0 to F7. The format in which numbers are stored in these registers is not specified. Floating point formats only become visible when a number is transferred to memory, using one of the precisions described below.

There is also a floating point status register. This is used to hold flags which indicate various error conditions, such as overflow and division by zero. Each flag has a corresponding mask, which can be used to enable or disable a 'trap' associated with the error condition.

Precision

All basic floating point instructions operate as though the result were computed to infinite precision and then rounded to the length and in the way specified by the instruction. The rounding is selectable from:

- Round to nearest
- Round to +infinity (P)
- Round to -infinity (M)
- Round to zero (Z).

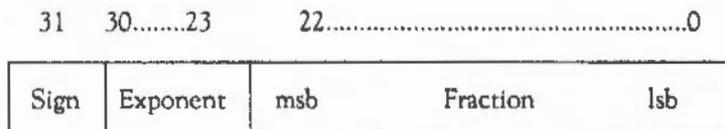
The default is 'round to nearest'. If any of the others is required they must be given in the instruction.

The working precision of the system is 80 bits, comprising a 64 bit mantissa, a 15 bit exponent and a sign bit.

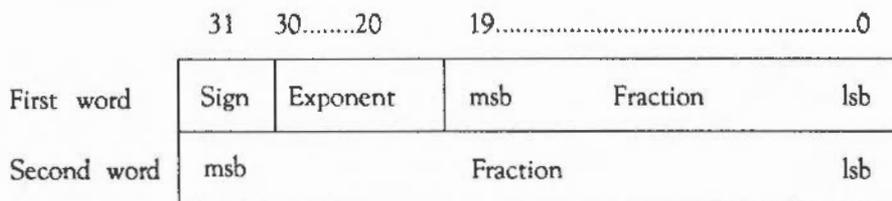
Like the ARM instructions, the floating point data processing operations refer to registers rather than memory locations. Values may be stored into ARM memory in one of four formats:

- The exponent uses excess n notation, where n is dependent on the format
- single and double precision have an implied 1 to the left of the binary point, except when the exponent is zero – at this point underflow starts to take over (see *UFL*)

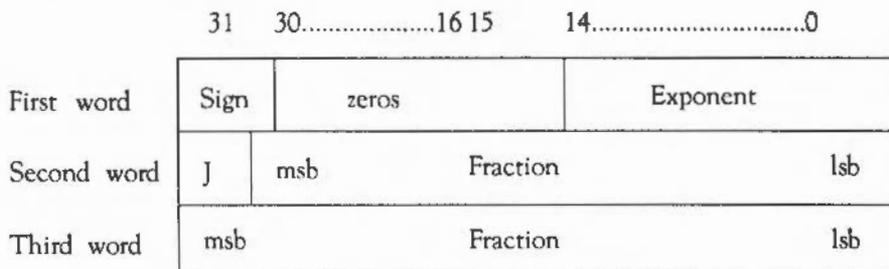
IEEE Single Precision (S)



IEEE Double Precision (D)



Double Extended Precision (E)



J is one bit to the left of the binary point

Storing a floating point register in 'E' format is guaranteed to maintain precision when loaded back into the floating point system in this format. However, note that the above layout will vary on future floating point systems, so software should not be written to depend on it.

Packed Decimal (P)

31.....0

First word	sign	e3	e2	e1	e0	d18	d17	d16
Second word	d15	d14	d13	d12	d11	d10	d9	d8
Third word	d7	d6	d5	d4	d3	d2	d1	d0

Value is :

$$\pm d * 10^{(\pm e)}$$

d18 or e3 is the most significant digit. Sign contains both the number's sign (top bit) and the exponent's sign (next bit). The other two bits are zero.

The value of 'd' is arranged with decimal point between d18 and d17 and is normalised so that for a normal number $1 \leq d18 \leq 9$. The guaranteed ranges for 'd' and 'e' are 17 digits and 3 digits respectively: e3 and d0, d1 may always be zero.

A single precision number has a maximum exponent of 53 and 9 digits of significance; a double precision number has a maximum exponent of 340 and 17 digits of significance. The result when the packed values are &A through &F is undefined. Zero will always be stored as +zero, but either +0 or -0 may be loaded.

Floating point status register

There is a floating point status register (FPSR) which, like ARM's combined PC and PSR, has all the necessary status for the floating point system. The FPSR contains the IEEE flags but not the result flags – these are only available after floating point compare operations.

Each IEEE flag denotes a possible error condition. There is a corresponding 'trap' or interrupt enable flag for each one. If the trap is enabled, then the error condition will cause execution to stop with an error; otherwise a special result (eg not-a-number or infinity) is returned.

	The flags contained in the status register are as follows:
IVO flag	<p>IVO – invalid operation</p> <p>The IVO is set when an operand is invalid for the operation to be performed. Invalid operations are:</p> <ul style="list-style-type: none"> • Any operation on something a NAN (not-a-number) • Magnitude subtraction of infinities eg +infinity + -infinity • Multiplication of 0 by an infinity • Division of 0/0 or infinity/infinity • $x \text{ REM } y$ where x is infinity or y is 0 • Square root of any number less than zero (but $\text{SQR}(-0)$ is -0) • Conversion to integer or decimal when overflow, infinity or operand not being a number make it impossible. • Comparison with exceptions of unordered operands. • ACS, ASN when argument's absolute value is > 1 • SIN, COS, TAN when argument is infinite • LOG, LGN when argument ≤ 0
REM flag	REM is the 'remainder after floating point division' operator.
DVZ flag	<p>DVZ – division by zero</p> <p>If the divisor is zero and the dividend a finite, non-zero number then this exception occurs, or a correctly signed infinity is returned if the trap is disabled.</p>
OFL flag	<p>OFL – overflow</p> <p>The OFL is set whenever the destination format's largest finite number is exceeded by the result after rounding has taken place. As overflow is detected after rounding a result, whether overflow occurs or not (after some operations) depends on rounding mode.</p>

The untrapped result returned is the correctly signed infinity, independent of the rounding mode – overflow can be seen as a signal that an infinite result has been generated from an operation on finite values.

UFL flag

UFL – underflow

The UFL is set whenever a result is so tiny that it is rounded to zero, but has a non-zero value. As underflow is detected after rounding a result, whether underflow occurs or not after some operations depends on rounding mode.

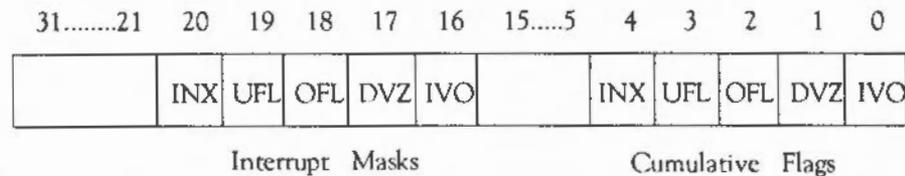
The untrapped result returned is zero, with the sign set to that of the non-zero value.

INX flag

INX – inexact

The INX is set if the rounded result of an operation is not exact (different from the value computable with infinite precision) or overflow has occurred while the OFL trap was disabled. If there is no trap the result will be used directly. OFL or UFL traps take precedence over INX. INX will also be set when computing SIN or COS or TAN of values larger than 10^{20} (ie values for which the multiple of PI ranging gives a useless answer). Different floating point implementations may vary in their use of the INX bit, so it is not recommended to construct software depending on this feature.

For each flag, there are two bits of the instruction dedicated to it:



Whenever the appropriate condition arises, the cumulative flags in bits 0 to 4 are set. They can only become cleared by a WFS instruction. If the relevant interrupt mask is set, then the same condition that sets the cumulative flags also causes an exception type error (bit 31 set) to be delivered to the program. The floating point system provides the exception routine with a word indicating (in the same position as the cumulative flags) which floating point exception occurred.

The instruction set

Co-Processor data transfer

`op<cond>prec Fd, addr`
`op` is LDF for load, STF for store
`addr` is `[Rn]<, #offset>` or `[Rn, #offset]<!>`
`prec` is the precision denoted by the letter S, D, E or P (see below)

The bit format of the instruction is:

31....28 27...24 23 22 21 20 19....16 15...12 11....8 7.....0

Cond	110P	U/D	Y	Wb	L/S	Rn	X Fd	0001	offset
------	------	-----	---	----	-----	----	------	------	--------

`P` is pre- or post-indexed addressing
`U/D` is positive/negative offset
`YX` is the precision
`Wb` is write-back (pre-indexed only)
`L/S` is load or store
`Rn` is the ARM base address register
`Fd` is the FPU source/destination register
`offset` is the scaled offset

Load (LDF) or store (STF) the high precision value into one of the four memory formats. On store, the value is rounded using the 'round to nearest' rounding method to the destination precision, or is precise if the destination has sufficient precision. Bits 22 and 15 are set from the precision letter, and determine the precision, as follows:

Precision	Letter	Y	X
Single	S	0	0
Double	D	0	1
Extended	E	1	0
Packed BCD	P	1	1

The `offset` is in words from the ARM base register, and is in the range `-1020` to `+1020`. It is added to the base register in pre-indexed mode if write-back is specified, and always in post-indexed mode.

Co-Processor data operations

The formats of these instructions are:

binop<cond>prec<round>Fd, Fn, (Fm | #value)
 unyop<cond>prec<round>Fd, (Fm | #value)

binop is one of the binary operations listed below
 unyop is one of the unary operations.
 Fd is the FPU destination register
 Fn is the FPU source register (binops only)
 Fm is the FPU source register
 #value is the immediate operand, as an alternative to Fm

The binops are:

			opcode
*ADF	Add:	$Fd := Fn + Fm$	00000
*MUF	Multiply:	$Fd := Fn * Fm$	00010
*SUF	Sub:	$Fd := Fn - Fm$	00100
*RSF	Reverse Subtract:	$Fd := Fm - Fn$	00110
*DVF	Divide:	$Fd := Fn / Fm$	01000
*RDF	Reverse Divide:	$Fd := Fm / Fn$	01010
POW	Power:	$Fd := Fn$ to the power of Fm	01100
RPW	Reverse Power:	$Fd := Fm$ to the power of Fn	01110
*RMF	Remainder	$Fd :=$ remainder of Fn / Fm	10000
*FML	Fast Multiply:	$Fd := Fn * Fm$	10010
*FDV	Fast Divide:	$Fd := Fn / Fm$	10100
*FRD	Fast Reverse Divide:	$Fd := Fm / Fn$	10110
POL	Polar angle (ArcTan2):	$Fd :=$ polar angle of (Fn, Fm)	11000

The unops are:

MVF	Move:	$Fd := Fm$	00001
MNF	Move Negated:	$Fd := -Fm$	00011
ABS	Absolute value:	$Fd := ABS (Fm)$	00101
RND	Round to integral value:	$Fd :=$ integer value of Fm	00111
*SQT	Square root:	$Fd :=$ square root of Fm	01001
LOG	Logarithm to base 10:	$Fd :=$ logten of Fm	01011
LGN	Logarithm to base e:	$Fd :=$ loge of Fm	01101
EXP	Exponent:	$Fd := e$ to the power of Fm	01111
*SIN	Sine:	$Fd :=$ sine of Fm	10001
*COS	Cosine:	$Fd :=$ cosine of Fm	10011
TAN	Tangent:	$Fd :=$ tangent of Fm	10101

ASN	Arc Sine:	Fd := arcsine of Fm	10111
ACS	Arc Cosine:	Fd := arccosine of Fm	11001
*ATN	Arc Tangent:	Fd := arctangent of Fm	11011

* Supported by floating point co-processor (if fitted)

Note that wherever Fm is mentioned, a floating point constant could be used instead.

FML, FRD and FDV produce a result only accurate to single precision.

Final rounding is done only at the last stage of a SIN, COS etc – the calculations to compute the value are done with 'round to nearest' using the full working precision.

The binary format of the instruction is:

31...28 27...24 23...20 19...16 15...12 11...8 7...4 3...0

Cond	1110	abcd	e Fn	j Fd	0001	fgh0	i Fm
------	------	------	------	------	------	------	------

abcdj is the opcode
 ef is the precision
 gh is the rounding mode
 i determines whether Fm is a register number or constant

Co-Processor register transfer

FLT<cond>prec<round>	Fn,Rd	0000
FIX<cond>prec<round>	Rd,Fm	0001
WFS<cond>	Rd	0010
RFS<cond>	Rd	0011
WFC<cond>	Rd	0100
RFC<cond>	Rd	0101

<round> is the optional rounding mode: P, M or Z; see below

FLT	Integer to Floating Point	Fn := Rd
FIX	Floating point to integer	Rd := Fm
WFS	Write Floating Point Status	FPSR := Rd

RFS	Read Floating Point Status	Rd := FPSR
WFC	Write Floating Point Control	FPC := Rd Supervisor Only
RFC	Read Floating Point Control	Rd := FPC Supervisor Only

The binary format is:

31...28	27...24	23...21	20	19...16	15...12	11...8	7...4	3...0
Cond	1110	abc	L/S	e Fn	Rd	0001	fgh1	i Fm

abcL/S are the operation code bits, as above (0110.. undefined)
 ef give the floating point precision, as above
 gh is the rounding mode, as below
 Fm, Fn are FPU register numbers
 Rd is an ARM register number
 i determines whether Fm is a register number or constant

The rounding modes are

Mode	Letter	g	h
Nearest		0	0
Plus infinity	P	0	1
Minus infinity	M	1	0
Zero	Z	1	1

The values allowed for immediate operands in Fm are:

Value	Fm encoding
0.0	000
1.0	001
2.0	010
3.0	011
4.0	100
5.0	101
0.5	110
10.0	111

Co-Processor Status Transfer

Constants cannot be specified in the Fm field for the FIX instruction since there is no point FIXing a known value into an ARM integer register. The MOV instruction should be used for this.

CMF<cond>prec<round>	Fm, Fn	100
CNF<cond>prec<round>	Fm, Fn	101
CMFE<cond>prec<round>	Fm, Fn	110
CNFE<cond>prec<round>	Fm, Fn	111

<round> is the optional rounding mode: P, M or Z (see below).

CMF	Compare floating	compare Fn with Fm
CNF	Compare negated floating	compare Fn with -Fm
CMFE	Compare floating with exception	compare Fn with Fm
CNFE	Compare negated floating with exception	compare Fn with -Fm

The binary format is:

31....28 27...24 23...21 20 19....16 15...12 11....8 7....4 3....0

Cond	1110	abc	1	e Fn	1111	0001	fgh1	i Fm
------	------	-----	---	------	------	------	------	------

- abcL/S are the operation code bits, as above (0110.. undefined)
- ef give the floating point precision, as above
- gh is the rounding mode, as below
- Fm, Fn are FPU register numbers
- i determines whether Fm is a register number or constant

The rounding modes are:

Mode	Letter	g	h
Nearest	0	0	
Plus infinity	P	0	1
Minus infinity	M	1	0
Zero	Z	1	1

FPEmulator_Version (SWI &40480)

SWI Calls

	Returns the version number of the floating point emulator
On entry	No parameters passed in registers
On exit	R0 = BCD version number
Interrupts	Interrupt status is undefined Fast interrupts are enabled
Processor mode	Processor is in SVC mode
Re-entrancy	Not defined
Use	This call returns the version number of the floating point emulator as a binary coded decimal (BCD) number in R0. This SWI will continue to be supported by the hardware expansion.
Related SWIs	None
Related vectors	None

ShellCLI

Introduction

This module provides a single *Command that allows you to invoke a command shell from a Wimp program.

It also has two SWIs for its own internal use. You must not use them in your own code.

* Commands

*ShellCLI

	Invoke a command shell from a Wimp program
Syntax	*ShellCLI
Parameters	None
Use	<p>This command is started as a Wimp task. It prompts the user with *, and passes the line typed to the command line interpreter, OS_CLI. This is repeated until the user enters a blank line, whereupon control is returned to the caller. The Task Manager uses this command to implement its *Command (F12) menu item.</p> <p>Notes: You must use Wimp_StartTask to call *ShellCLI, not OS_CLI. The command uses the two SWIs Shell_Create and Shell_Destroy; it is the only user of these SWIs. Note that you can only call Wimp_StartTask or *WimpTask from within an active task.</p>
Example	*wimpTask ShellCLI
Related commands	None
Related SWIs	Shell_Create, Shell_Destroy
Related vectors	None

SWI Calls

Shell_Create (SWI &405C0)

This SWI call is for use by the ShellCLI module only. You must not use it in your own code.

Shell_Destroy (SWI &405C1)

This SWI call is for use by the ShellCLI module only. You must not use it in your own code

Command scripts

Introduction

Command scripts are files of commands that you would normally type in at the Command Line prompt. There are two common reasons for using such a file:

- To set up the computer to the state you want, either when you switch on or when you start an application.
- To save typing in a set of commands you find yourself frequently using.

In the first case the file of commands is commonly known as a boot file.

You may find using an Alias\$... variable to be better in some cases. The main advantage of these variables is that they are held in memory and so are quicker in execution; however, they are only really suitable for short commands. Even if you use these variables you are still likely to need to use a command file to set them up initially.

There are two types of file available for writing command scripts: Command files, and Obey files. The differences between these two file types are:

- An Obey file is read directly, whereas a Command file is treated as if it were typed at the keyboard (and hence usually appears on the screen).
- An Obey file sets the system variable Obey\$Dir to the directory it is in.
- An Obey file can be passed parameters
- An Obey file stops when an error is returned to the Obey module (or when an error is generated and the exit handler is the Obey module – an untrapped error, not in an application).

Overview and Technical Details

Creating a command script

A command script can be created using any text or word processor. Normally you then have to use the command `*SetType` to set the type of the file to `Command` or `Obey`.

You should save it in one of the following:

- the directory from which the command script will be run (typically your root directory, or an application directory)
- the library (typically `$.Library`, but may be `$.ArthurLib` on a network; see `*Configure Lib` in the chapter entitled *FileSwitch*).

Running the script

Provided that you have set the file to have a filetype of `Command` or `Obey` it can then be run in the same ways as any other file:

- Type its name at the `*` prompt.
- Type its name preceded by a `*` at any other prompt (some applications may not support this).
- Double-click on its icon from the desktop.

The same restrictions apply as with any other file. If the file is not in either your current directory or the library, it will not be found if you just give the filename; you must give its full pathname. (This assumes you have not changed the value of the system variable `Run$Path`.)

You can force any text file to be treated as an obey file by using the command `*Obey`. This overrides the current file type, such as `Text` or `Command`. Obviously, this will only have meaning if the text in the file is valid to treat as an obey file.

Similarly, any file can be forced to be a command file by using `*Exec`. This is described in the chapter entitled *Character Input*.

Obey\$Dir

When an obey file is run, by using any of the above techniques, the system variable `Obey$Dir` is set to the parent directory part of the pathname used. For example, if you were to type `*Obey a.b.c`, then `a.b` is the parent directory of the pathname.

Note that it is not set to the full parent name, only the part of the string passed to the command as the pathname. So if you change the current directory or filing system during the obey file, then it would not be valid any more.

Ideally, you should invoke Obey files (and applications, which are started by an Obey file named !Run) by using their full pathname, and preceding that by either a forward slash / or the word Run, for example:

```
/ adfs::MikeWinnie.$Odds'nSods.MyConfig
Run adfs::MikeWinnie.$Odds'nSods.MyConfig
```

This ensures that Obey\$Dir is set to the full pathname of the Obey file.

Run\$Path

The variable Run\$Path also influences how this parent name is decoded. If you were to type:

```
*Set Run$Path adfs::Winnie.Flagstaff.
*obeyfile par1 par2
```

Then it would be interpreted as:

```
*Run adfs::Winnie.Flagstaff.obeyfile par1 par2
```

If the filetype of obeyfile was &FEB, an obey file, then the command would be interpreted as:

```
*Obey adfs::Winnie.Flagstaff.obeyfile par1 par2
```

This can also apply to application directories, as follows:

```
*Set Alias$@RunType_FEB Obey %*0
*Set File$Type_FEB Obey
*Set Run$Path adfs::Winnie.Flagstaff.
*appdir par1 par2
```

In this case, RISC OS would look for the !Run file within the application directory and run it. Note that in most cases, the first two lines above are already defined in your system. If !Run is an obey file, then it would be interpreted as:

```
*Obey adfs::Winnie.Flagstaff.appdir.!Run par1 par2
```

Making a script run automatically

Note that Obey files can also be nested to refer to other files to Obey; however, Command files cannot be nested. This is one of the reasons why it is better to set up your file as an Obey file rather than a Command file

You can make scripts run automatically:

- From the network when you first log on.

The file must be called !ArmBoot. (This is to distinguish a boot file for a machine running Arthur or RISC OS from an existing !Boot file already on the network for the use of BBC model A, model B or Master series computers.

- From a disc when you first switch the computer on.

The file must be called !Boot.

- From an application directory when you first display the directory's icon under the desktop.

The file must be called !Boot. It is run if RISCOS does not already know of a sprite having the same name as the directory, and is intended to load sprites for applications when they first need to be displayed. For further details see the chapter entitled *The Window Manager*.

- From an application directory when the application is run.

The file must be called !Run. For further details see the chapter entitled *The Window Manager*.

In the first two cases you will need to use the *Opt command as well.

For an example of the latter two cases, you can look in any of the application directories in the Applications Suite. If you are using the desktop, you will need to hold down the Shift key while you open the application directory, otherwise the application will run.

Using parameters

An Obey file can have parameters passed to it, which can then be used by the command script. A Command file cannot have parameters passed to it. The first parameter is referred to as %0, the second as %1, and so on. You can refer to all the parameters after a particular one by putting a * after the %, so %*1 would refer to the all parameters from the second one onwards.

These parameters are substituted before the line is passed to the Command Line interpreter. Thus if an Obey file called Display contained:

```
FileInfo %0  
Type %0
```

then the command *Display MyFile would do this:

```
FileInfo MyFile  
Type MyFile
```

Sometimes you do not want parameter substitution. For example, suppose you wish to include a *Set Alias\$. command in your file, such as:

```
Set Alias$Mode echo |<22>|<%0>          Desired command
```

The effect of this is to create a new command 'Mode'. If you include the *Set Alias command in an Obey file, when you run the file the %0 will be replaced by the first parameter passed to the file. To prevent the substitution you need to change the % to %%:

```
Set Alias$Mode echo |<22>|<%%0>          Command needed in file
```

Now when the file is run, the '%%0' is changed to '%0'. No other substitution occurs at this stage, and the desired command is issued. See the *Set command in the chapter entitled *Program Environment*.

*Obey

*Commands

Executes a file of * commands

Syntax

```
*Obey [<pathname> [<parameters>]]
```

Parameters

<pathname> a valid pathname, specifying a file
<parameters> strings separated by spaces

Use

*Obey executes a file of * commands. Argument substitution is performed on each line, using parameters passed in the command.

Example

```
*Obey !commands myfile1 12
```

Related commands

*Exec

Related SWIs

None

Related vectors

None

Application Notes

These example files illustrate several of the important differences between Command and Obey files:

```
*BASIC
AUTO
FOR I = 1 TO 10
  PRINT "Hello"
NEXT I
END
```

If this were a command file, it would enter the BASIC interpreter, and input the file shown. The command script will end with the BASIC interpreter waiting for another line of input. You can then press Esc to get a prompt, type RUN to run the program, and then type QUIT to leave BASIC. This script shows how a command file is passed to the input, and can change what is accepting its input (in this case to the BASIC interpreter).

In contrast, if this were an Obey file it would be passed to the Command Line interpreter, and an attempt would be made to run these commands:

```
*BASIC
*AUTO
*FOR I = 1 TO 10
*  PRINT "Hello"
*NEXT I
*END
```

Only the first command is valid, and so as an Obey file all this does is to leave you in the BASIC interpreter. Type QUIT to leave BASIC; you will then get an error message saying File 'AUTO' not found, generated by the second line in the file.

The next example illustrates how control characters are handled:

```
echo <?>
echo |<?>
```

The control characters are represented in GSTrans format (see the chapter entitled *Conversions*). These are not interpreted until the echo command is run, and are only interpreted then because echo expects GSTrans format.

The first line sends an ASCII 7 to the VDU drivers, sounding a beep; see the chapter entitled *VDU drivers* for more information. In the second line, the `|` preceding the `<` changes it from the start of a GSTrans sequence to just representing the character `<`, so the overall effect is:

```
echo <7>          Send ASCII 7 to VDU drivers – beeps
```

```
echo |<7>        Send <7> to the screen
```

The last examples are a Command file:

```
*Set Alias$more %echo |<14>|m %type -tabexpand %%*0|m %echo |<15>
```

and an Obey file that has the same effect:

```
Set Alias$more %echo |<14>|m %type -tabexpand %%*0|m %echo |<15>
```

The only differences between the two examples are that the Command file has a preceding `*` added, to ensure that the command is passed to the Command Line interpreter; and that the Obey file has the `%*0` changed to `%%*0` to delay the substitution of parameters.

The file creates a new command called 'more' – taking its name from the Unix 'more' command – by setting the variable `Alias$more`:

- The `%` characters that precede `echo` and `type` ensure that the actual commands are used, rather than an aliased version of them.
- The sequence `|m` represents a carriage return in GSTrans format and is used to separate the commands, just as Return would if you were typing the commands.
- The two `echo` commands turn paged mode on, then off, by sending the control characters ASCII 14 and 15 respectively to the VDU drivers (see the chapter entitled *VDU drivers* for more information).
- The `|` before each `<` prevents the control characters from being interpreted until the aliased command 'more' is run.

The command turns paged mode on, types a file to the screen expanding tabs as it does so, and then turns paged mode off.

Appendices and Tables

Appendix A - ARM assembler

Introduction

Assembly language is a programming language in which each statement translates directly into a single machine code instruction or piece of data. An assembler is a piece of software which converts these statements into their machine code counterparts.

Writing in assembly language has its disadvantages. The code is more verbose than the equivalent high-level language statements, more difficult to understand and therefore harder to debug. High-level languages were invented so that programs could be written to look more like English so we could talk to computers in our language rather than directly in its own.

There are two reasons why, in certain circumstances, assembly language is used in preference to high-level languages. The first reason is that the machine code program produced by it executes more quickly than its high-level counterparts, particularly those in languages such as BASIC which are interpreted. The second reason is that assembly language offers greater flexibility. It allows certain operating system routines to be called or replaced by new pieces of code, and it allows greater access to the hardware devices and controllers.

Using the BASIC assembler

The assembler is part of the BBC BASIC language. Square brackets '[' and ']' are used to enclose all the assembly language instructions and directives and hence to inform BASIC that the enclosed instructions are intended for its assembler. However, there are several operations which must be performed from BASIC itself to ensure that a subsequent assembly language routine is assembled correctly.

Initialising external variables

The assembler allows the use of BASIC variables as addresses or data in instructions and assembler directives. For example variables can be set up in BASIC giving the numbers of any SWI routines which will be called. For example:

```

OS_WriteI = &100
.....
[
.....
SWI OS_WriteI+ASC">"
.....

```

Reserving memory space
for the machine code

The machine code generated by the assembler is stored in memory. However, the assembler does not automatically set memory aside for this purpose. You must reserve sufficient memory by using the DIM statement. For example:

```
1000 DIM code% 100
```

The start address of the memory area reserved is assigned to the variable code%. The address of the last memory location is code%+100. Hence, it reserves a total of 101 bytes of memory.

Memory pointers

You need to tell the assembler the start address of the area of memory you have reserved. The simplest way to do this is to assign P% to point to the start of this area. For example:

```

DIM code% 100
.....
P% = code%

```

P% is then used as the program counter. The assembler places the first assembler instruction at the address P% and automatically increments the value of P% by four so that it points to the next free location. When the assembler has finished assembling the code, P% points to the byte following the final location used. Therefore, the number of bytes of machine code generated is given by:

```
P% - code%
```

This method assumes that you wish subsequently to execute the code at the same location.

The position in memory at which you load a machine code program may be significant. For example, it might refer directly to data embedded within itself, or expect to find routines at fixed addresses. Such a program only works if it is loaded in the correct place in memory. However, it is often inconvenient to assemble the program directly into the place where it will

Implementing passes

eventually be executed. This memory may well be used for something else whilst you are assembling the program. The solution to this problem is to use a technique called 'offset assembly' where code is assembled as if it is to run at a certain address but is actually placed at another.

To do this, set O% to point to the place where the first machine code instruction is to be placed and P% to point to the address where the code is to be run.

To notify the assembler that this method of generating code is to be used, the directive OPT, which is described in more detail below, must have bit 2 set.

It is usually easy, and always preferable, to write ARM code that is position independent.

Normally, when the processor is executing a machine code program, it executes one instruction and then moves on automatically to the one following it in memory. You can, however, make the processor move to a different location and start processing from there instead by using one of the 'branch' instructions. For example:

```
.result_was_0
.....
BEQ result_was_0
```

The fullstop in front of the name result_was_0 identifies this string as the name of a 'label'. This is a directive to the assembler which tells it to assign the current value of the program counter (P%) to the variable whose name follows the fullstop.

BEQ means 'branch if the result of the last calculation that updated the PSR was zero'. The location to be branched to is given by the value previously assigned to the label result_was_0.

The label can, however, occur after the branch instruction. This causes a slight problem for the assembler since when it reaches the branch instruction, it hasn't yet assigned a value to the variable, so it doesn't know which value to replace it with.

You can get around this problem by assembling the source code twice. This is known as two-pass assembly. During the first pass the assembler assigns values to all the label variables. In the second pass it is able to replace references to these variables by their values.

It is only when the text contains no forward references of labels that just a single pass is sufficient.

These two passes may be performed by a FOR...NEXT loop as follows:

```
DIM code% 400
FOR pass% = 0 TO 3 STEP 3
  P% = code%
  [
  OPT pass%
  ...
  ]
NEXT pass%
```

Note that the pointer(s), in this case just P%, must be set at the start of both passes.

The OPT directive

The OPT is an assembler directive whose bits have the following meaning:

Bit Meaning

- 0 Assembly listing enabled if set
- 1 Assembler errors enabled
- 2 Assembled code placed in memory at O% instead of P%
- 3 Check that assembled code does exceed memory limit L%

Bit 0 controls whether a listing is produced. It is up to you whether or not you wish to have one or not.

Bit 1 determines whether or not assembler errors are to be flagged or suppressed. For the first pass, bit 1 should be zero since otherwise any forward-referenced labels will cause the error `Unknown or missing variable` and hence stop the assembly. During the second pass, this bit should be set to one, since by this stage all the labels defined are known, so the only errors it catches are 'real ones' – such as labels which have been used but not defined.

Bit 2 allows 'offset assembly', ie the program may be assembled into one area of memory, pointed to by O%, whilst being set up to run at the address pointed to by P%.

Bit 3 checks that the assembled code does not exceed the area of memory that it has been reserved. Its normal usage is:

```
DIM code% 10000, L%-1  
...
```

Executing a machine code program

Once an assembly language routine has been successfully assembled, the resulting machine code can be executed in a variety of ways:

```
CALL <address>  
USR <address>
```

These may be used from inside BASIC to run the machine code at a given address. See the *BBC Basic Guide* for more details on these statements.

The commands below will load and run the named file, using either its filetype (such as &FF8 for absolute code) and their associated system variables, or the load and execution addresses defined when it was saved.

```
*<name>  
*RUN <name>  
*/<name>
```

We strongly advise you to use file types in preference to load and execution addresses.

Format of assembly language statements

The assembly language statements and assembler directives should be between the square brackets.

There are very few rules about the format of assembly language statements, those which exist are given below:

- Each assembly language statement comprises an assembler mnemonic of one or more letters followed by a varying number of operands.
- Instructions should be separated from each other by colons or newlines.
- Any text following a full stop '.' is treated as a label name.

- Any text following a semicolon ';', or backslash '\', or 'REM' is treated as a comment and so ignored (until the next end of line or ':').
- Spaces between the mnemonic and the first operand, and between the operands themselves are ignored.

The BASIC assembler contains the following directives:

EQUB int	Define 1 byte of memory from LSB of int (DCB,=)
EQUW int	Define 2 bytes of memory from int (DCW)
EQU4 int	Define 4 bytes of memory from int (DCD)
EQU5 str	Define 0 - 255 bytes as required by string expression str (DCS)
ALIGN	Align P% (and O%) to the next word boundary
ADR reg,addr	Assemble instruction to load addr into reg

- The first four operations initialise the reserved memory to the values specified by the operand. In the case of EQU5 the operand field should be a string expression. In all other cases it may be a numeric expression. DCB, DCW, DCD and DCS are synonyms for these directives.
- The ALIGN directive ensures that the next P% (and O%) that is used lies on a word boundary. It is used after, for example, an EQU5 to ensure that the next instruction is word-aligned.

Note that although instructions are word-aligned, labels are not. So:

```

ALIGN
EQUB 0
.label SUBS r0, r0, #1
      B label

```

is different to:

```

ALIGN
EQUB 0
.label SUBS r0, r0, #1 ; B will jump to EQUB and the 3 undefined
      B label ; words, not label!

```

- ADR assembles a single instruction, an ADD or SUB, with reg as the destination register. It obtains addr in that register in a PC-relative (ie position independent) manner.

Registers

At any particular time there are sixteen 32-bit registers available for use, R0 to R15. However, R15 is special since it contains the program counter and the processor status register.

R15 is split up with 24 bits used as the program counter (PC) to hold the word address of the next instruction. 8 bits are used as the processor status register (PSR) to hold information about the current values of flags and the current mode/register bank. These bits are arranged as follows:

The top six bits hold the following information:

Bit	Flag	Meaning
31	N	Negative flag
30	Z	Zero flag
29	C	Carry flag
28	V	Overflow flag
27	I	Interrupt request disable
26	F	Fast interrupt request disable

The bottom two bits can hold one of four different values:

M	Meaning
0	User mode
1	Fast interrupt processing mode (FIQ mode)
2	Interrupt processing mode (IRQ mode)
3	Supervisor mode (SVC mode)

User mode is the normal program execution state. SVC mode is a special mode which is entered when calls to the supervisor are made using software interrupts (SWIs) or when an exception occurs. From within SVC mode certain operations can be performed which are not permitted in user mode, such as writing to hardware devices and peripherals. SVC mode has its own private registers R13 and R14. So after changing to SVC mode, the registers R0 - R12 are the same, but new versions of R13 and R14 are available. The values contained by these registers in user mode are not overwritten or corrupted.

Similarly, IRQ and FIQ modes have their own private registers (R13 - R14 and R8 - R14 respectively).

Although only 16 registers are available at any one time, the processor actually contains a total of 27 registers.

For a more complete description of the registers, see the chapter entitled *ARM Hardware*.

Condition codes

All the machine code instructions can be performed conditionally according to the status of one or more of the following flags: N, Z, C, V. The sixteen available condition codes are:

AL	Always	<i>This is the default</i>
CC	Carry clear	C clear
CS	Carry set	C set
EQ	Equal	Z set
GE	Greater than or equal	(N set and V set) or (N clear and V clear)
GT	Greater than	((N set and V set) or (N clear and V clear)) and Z clear
HI	Higher (unsigned)	C set and Z clear
LE	Less than or equal	(N set and V clear) or (N clear and V set) or Z set
LS	Lower or same (unsigned)	C clear or Z set
LT	Less than	(N set and V clear) or (N clear and V set)
MI	Negative	N set
NE	Not equal	Z clear
NV	Never	
PL	Positive	N clear
VC	Overflow clear	V clear
VS	Overflow set	V set

Two of these may be given alternative names as follows:

LO	Lower unsigned	is equivalent to CC
HS	Higher / same unsigned	is equivalent to CS

The instruction set

The available instructions are introduced below in categories indicating the type of action they perform and their syntax. The description of the syntax obeys the following standards:

« »	indicates that the contents of the brackets are optional (unlike all other chapters, where we have been using [] instead)
(x y)	indicates the either x or y but not both may be given
#exp	indicates that an expression is to be used which evaluates to an immediate constant. An error is given if the value cannot be stored in the instruction.
Rn	indicates that an expression evaluating to a register number (in the range 0-15) should be used, or just a register name, eg R0. PC may be used for R15.
shift	indicates that one of the following shift options should be used: ASL (Rn #exp) Arithmetic shift left by contents of Rn or expression LSL (Rn #exp) Logical shift left ASR (Rn #exp) Arithmetic shift right LSR (Rn #exp) Logical shift right ROR (Rn #exp) Rotate right RRX Rotate right one bit with extend

In fact ASL and LSL are the same. The ARM does not provide true arithmetic shifts. LSL is the preferred form, as it indicates this.

Syntax:

opcode«cond»«S» Rd, «Rn», (#exp | Rm «,shift»)

The instructions available are given below:

Instruction		Calculation performed
ADC	Add with carry	$Rd = Rn + Rm + C$
ADD	Add without carry	$Rd = Rn + Rm$
SBC	Subtract with carry	$Rd = Rn - Rm - (1 - C)$
SUB	Subtract without carry	$Rd = Rn - Rm$
RSC	Reverse subtract with carry	$Rd = Rm - Rn - (1 - C)$
RSB	Reverse subtract without carry	$Rd = Rm - Rn$
AND	Bitwise AND	$Rd = Rn \text{ AND } Rm$
BIC	Bitwise AND NOT	$Rd = Rn \text{ AND NOT } (Rm)$
ORR	Bitwise OR	$Rd = Rn \text{ OR } Rm$
EOR	Bitwise EOR	$Rd = Rn \text{ EOR } Rm$
MOV	Move	$Rd = Rm$
MVN	Move NOT	$Rd = \text{NOT } Rm$

Each of these instructions produces a result which it places in a destination register (Rd). The instructions do not affect bytes in memory directly.

As was seen above, all of these instructions can be performed conditionally. In addition, if the 'S' is present, they can cause the condition codes to be set or cleared. The condition codes N, Z, C and V are set by the arithmetic logic unit (ALU) in the arithmetic operations. The logical (bitwise) operations set N and Z from the ALU, C from the shifter (but only if it is used), and do not affect V.

Examples:

```
ADDEQ R1, R1, #7      ; If the zero flag is set then add 7
                    ; to the contents of register R1.

SBCS R2, R3, R4      ; Subtract with carry the contents of register R4
                    ; from the contents of register R3 and place the result
                    ; in register R2. The flags will be updated.

AND R3, R1, R2, LSR #2 ; Perform a logical AND on the contents of register R1
                    ; and the contents of register R2 * 4, and place the
                    ; result in register R3.
```

Comparisons

Special actions are taken if any of the source registers are R15; the action is as follows:

- If $R_m=R15$ all 32 bits of R15 are used in the operation ie the PC + PSR.
- If $R_n=R15$ only the 24 bits of the PC are used in the operation.

If the destination register is R15, then the action depends on whether the optional 'S' has been used:

- If S is not present only the 24 bits of the PC are set.
- If S is present the whole result is written to R15, the flags are updated from the result. (However the mode, I and F bits can only be changed when in non-user modes.)

Syntax:

opcode<cond>*S|P* Rn, (#exp|Rm <,shift>)

There are four comparison instructions:

Instruction		Calculation performed
CMN	Compare	$R_n + R_m$
CMP	Compare	$R_n - R_m$
TEQ	Test equal	$R_n \text{ EOR } R_m$
TST	Test	$R_n \text{ AND } R_m$

These are similar to the arithmetic and logical instructions listed above except that they do not take a destination register since they do not return a result. Also, they automatically set the condition flags (since they would perform no useful purpose if they didn't). Hence, the 'S' of the arithmetic instructions is implied. You can put an 'S' after the instruction to make this clearer.

These routines have an additional function which is to set the whole of the PSR to a given value. This is done by using a 'P' after the opt code, for example TEQP.

Normally the flags are set depending on the value of the comparison. The I and F bits and the mode and register bits are unaltered. The 'P' option allows the corresponding eight bits of the result of the calculation performed by the comparison to overwrite those in the PSR (or just the flag bits in user mode).

Example

```
TEQP    PC, #0x80000000    ; Set N flag, clear all others. Also enable
                               ; IRQs, FIQs, select User mode if privileged
```

The above example (as well as setting the N flag and clearing the others) will alter the IRQ, FIQ and mode bits of the PSR – but only if you are in a privileged mode.

The 'P' option is also useful in user mode, for example to collect errors:

```
STMFD   sp!, {r0, r1, r14}
...
BL      routine1
STRVS   r0, [sp, #0]        ; save error block pointer in return r0
                               ; in stack frame if error
MOV     r1, pc              ; save psr flags in r1
BL      routine2           ; called even if error from routine1
STRVS   r0, [sp, #0]        ; to do some tidy up action etc.
TEQVCP  r1, #0             ; if routine2 didn't give error, restore
LDMFD   sp!, {r0, r1, pc}   ; error indication from r1
```

Multiply instructions

Syntax:

```
MUL <cond> <S> Rd,Rm,Rs
MLA <cond> <S> Rd,Rm,Rs,Rn
```

There are two multiply instructions:

Instruction		Calculation performed
MUL	Multiply	$Rd = Rm * Rs$
MLA	Multiply-accumulate	$Rd = Rm * Rs + Rn$

The multiply instructions perform integer multiplication, giving the least significant 32 bits of the product of two 32-bit operands.

The destination register must not be R15 or the same as Rm. Any other register combinations can be used.

If the 'S' is given in the instruction, the N and Z flags are set on the result, and the C and V flags are undefined.

Branching instructions

Examples:

```
MUL    R1,R2,R3
MLAEQS R1,R2,R3,R4
```

Syntax:

```
B«cond» expression
BL«cond» expression
```

There are essentially only two branch instructions but in each case the branch can take place as a result of any of the 16 condition codes:

Instruction

```
B«cond»   Branch
BL«cond»  Branch and link
```

The branch instruction causes the execution of the code to jump to the instruction given at the address to be branched to. This address is held relative to the current location.

Example:

```
BEQ label1 ; branch if zero flag set
BMI minus  ; branch if negative flag set
```

The branch and link instruction performs the additional action of copying the address of the instruction following the branch, and the current flags, into register R14. R14 is known as the 'link register'. This means that the routine branched to can be returned from by transferring the contents of R14 into the program counter and can restore the flags from this register on return. Hence instead of being a simple branch the instruction acts like a subroutine call.

Example:

```
BLEQ equal
      .....; address of this instruction
      .....; moved to R14 automatically

.equal .....; start of subroutine
      .....

MOVS R15,R14 ; end of subroutine
```

Syntax:

opcode«cond»«B»«T» Rd, address

The single register load/save instructions are as follows:

Instruction

LDR	Load register
STR	Store register

These instructions allow a single register to load a value from memory or save a value to memory at a given address.

The instruction has two possible forms:

- the address is specified by register(s), whose names are enclosed in square brackets
- the address is specified by an expression

Address given by registers

The simplest form of address is a register number, in which case the contents of the register are used as the address to load from or save to. There are two other alternatives:

- pre-indexed addressing (with optional write back)
- post-indexed addressing (always with write back)

With pre-indexed addressing the contents of another register, or an immediate value, is added to the contents of the first register. This sum is then used as the address. It is known as pre-indexed addressing because the address being used is calculated before the load/save takes place. The first register (Rn below) can be optionally updated to contain the address which was actually used by adding a '!' after the closing square bracket.

Address syntax	Address
[Rn]	Contents of Rn
[Rn,#m]«!»	Contents of Rn + m
[Rn,Rm]«!»	Contents of Rn + contents of Rm
[Rn,Rm,shift #s]«!»	Contents of Rn + (contents of Rm shifted by s places)

With post-indexed addressing the address being used is given solely by the contents of the register Rn. The rest of the instruction determines what value is written back into Rn. This write back is performed automatically; no '!' is needed. Post-indexing gets its name from the fact that the address that is written back to Rn is calculated after the load/save takes place.

Address syntax	Value written back
[Rn],#m	Contents of Rn + m
[Rn],Rm	Contents of Rn + contents of Rm
[Rn],Rm,shift #s	Contents of Rn + (contents of Rm shifted by s places)

Address given as an expression

If the address is given as a simple expression, the assembler will generate a pre-indexed instruction using R15 (the PC) as the base register. If the address is out of the range of the instruction (4095 bytes), an error is given.

Options

If the 'B' option is specified after the condition, only a single byte is transferred, instead of a whole word. The top 3 bytes of the destination register are cleared by an LDRB instruction.

If the 'T' option is specified after the condition, then the TRAns pin on the ARM processor will be active during the transfer, forcing an address translation. This allows you to access User mode memory from a privileged mode. This option is invalid for pre-indexed addressing.

Using the program counter

If you use the program counter (PC, or R15) as one of the registers, a number of special cases apply:

- the PSR is never modified, even when Rd or Rn is the PC
- the PSR flags are not used when the PC is used as Rn, and (because of pipelining) it will be advanced by eight bytes from the current instruction
- the PSR flags are used when the PC is used as Rm, the offset register.

Syntax:

opcode<cond>type Rn<!,>, {Rlist}<^>

These instructions allow the loading or saving of several registers:

Instruction

LDM	Load multiple registers
STM	Store multiple registers

The contents of register Rn give the base address from/to which the value(s) are loaded or saved. This base address is effectively updated during the transfer, but is only written back to if you follow it with a '!'.
Rlist provides a list of registers which are to be loaded from or saved to. The order the registers are given, in the list, is irrelevant since the lowest numbered register will be loaded first and the highest one last. For example, a list comprising {R5,R3,R1,R8} will be loaded from/saved to in the order R1, R3, R5, R8, with R1 occupying the lowest address in memory. You can specify consecutive registers as a range; so {R0-R3} and {R0,R1,R2,R3} are equivalent.

The type is a two character mnemonic specifying either how Rn is updated, or what sort of a stack results:

Mnemonic	Meaning
DA	Decrement Rn After each store/load
DB	Decrement Rn Before each store/load
IA	Increment Rn After each store/load
IB	Increment Rn Before each store/load
EA	Empty Ascending stack is used
ED	Empty Descending stack is used
FA	Full Ascending stack is used
FD	Full Descending stack is used

- an empty stack is one in which the stack pointer points to the first free slot in it
- a full stack is one in which the stack pointer points to the last data item written to it

- an ascending stack is one which grows from low memory addresses to high ones
- a descending stack is one which grows from high memory addresses to low ones

In fact these are just different ways of looking at the situation – the way Rn is updated governs what sort of stack results, and vice versa. So, for each type of instruction in the first group there is an equivalent in the second:

LDMEA	is the same as	LDMDB
LDMED	is the same as	LDMIB
LDMFA	is the same as	LMDMA
LDMFD	is the same as	LDMIA
STMEA	is the same as	STMIA
STMED	is the same as	STMDA
STMFA	is the same as	STMIB
STMFD	is the same as	STMDB

All Acorn software uses an FD (full, descending) stack. If you are writing code for SVC mode you should try to also use a full descending stack – although you can use any type you like.

A '^' at the end of the register list has two possible meanings:

- For a load with R15 in the list, the '^' forces update of the PSR.
- Otherwise the '^' forces the load/store to access the User mode registers. The base is still taken from the current bank though, and if you try to write back the base it will be put in the User bank – probably not what you would have intended.

Examples:

```

LDMIA R5, {R0,R1,R2}           ; where R5 contains the value
                                ; 1484
                                ; This will load R0 from 1484
                                ;           R1 from 1488
                                ;           R2 from 148C

LDMDB R5, {R0-R2}             ; where R5 contains the value
                                ; 1484
                                ; This will load R0 from 1480
                                ;           R1 from 147C
                                ;           R2 from 1478

```

If there were a '!' after R5, so that it were written back to, then this would leave R5 containing &1490 and &1478 after the first and second examples respectively.

The examples below show directly equivalent ways of implementing a full descending stack. The first uses mnemonics describing how the stack pointer is handled:

```
STMDB Stackpointer!, {R0-R3}      ; push onto stack
LDMIA Stackpointer!, {R0-R3}      ; pull from stack
```

and the second uses mnemonics describing how the stack behaves:

```
STMFDB Stackpointer!, {R0,R1,R2,R3} ; push onto stack
LDMFDB Stackpointer!, {R0,R1,R2,R3} ; pull from stack
```

Using the base register

- You can always load the base register without any side effects on the rest of the LDM operation, because the ARM uses an internal copy of the base, and so will not be aware that it has been loaded with a new value.
- You can store to the base register as well. If you are not using write back then no problem will occur. If you are, then this is the order in which the ARM does the STM:
 - 1 write the lowest numbered register to memory
 - 2 do the write back
 - 3 write the other registers to memory in ascending order.

So, if the base register is the lowest-numbered one in the list, its original value is stored:

```
STMIA R2!, {R2-R6}      ; R2 stored is value before write back
```

Otherwise its written back value is stored:

```
STMIA R2!, {R1-R5}      ; R2 stored is value after write back
```

Using the program counter

If you use the program counter (PC, or R15) in the list of registers:

- the PSR is saved with the PC; and (because of pipelining) it will be advanced by twelve bytes from the current position
- the PSR is only loaded if you follow the register list with a '^'; and even then, only the bits you can modify in the ARM's current mode are loaded.

It is generally not sensible to use the PC as the base register. If you do:

- the PSR bits are used as part of the address, which will give an address exception unless all the flags are clear and all interrupts are enabled
- write back is switched off.

SWI instruction

Syntax:

SWI«cond» «expression»

SWI«cond» «"SWIname"» (BBC BASIC)

The SWI mnemonic stands for SoftWare Interrupt. On encountering a SWI, the ARM processor changes into SVC mode and stores the address of the next location in R14_svc – so the User mode value of R14 is not corrupted. The ARM then goes to the SWI routine handler via the hardware SWI vector containing its address.

The first thing that this routine does is to discover which SWI was requested. It finds this out by using the location addressed by (R14_svc – 4) to read the current SWI instruction. The opcode for a SWI is 32 bits long; 4 bits identify the opcode as being for a SWI, 4 bits hold all the condition codes and the bottom 24 bits identify which SWI it is. Hence 2^{24} different SWI routines can be distinguished.

When it has found which particular SWI it is, the routine executes the appropriate code to deal with it and then returns by placing the contents of R14_svc back into the PC, which restores the mode the caller was in.

This means that R14_svc will be corrupted if you execute a SWI in SVC mode – which can have disastrous consequences unless you take precautions.

The most common way to call this instruction is by using the SWI name, and letting the assembler translate this to a SWI number. The BBC BASIC assembler can do this translation directly:

```
SWINE "OS_WriteC"
```

See the chapter entitled *Introduction to SWIs* for a full description of how RISC OS handles SWIs, and the index of SWIs for a full list of the operating system SWIs.

Appendix B - Linker

Introduction

The ARM linker (Link) accepts as input one or more object files and object libraries written in the following formats:

- ARM Object Object (AOF)
- ARM Library Format (ALF)
- Acorn Unix a.out format
- Acorn Unix ar format

Object libraries (ALF, ar format) must include a symbol table as generated by ObjLib (for AOF/ALF) or ranlib (for a.out/ar).

In general, the mixing of AOF and a.out is not encouraged – it can be made to work, but with restrictions.

Each object file and object library member consists of some number of *areas*. Very often, an AOF object contains a *code* area, a *data* area and a *debug* area (more are possible – see the appendix entitled *ARM Object Format*). An a.out object, on the other hand, always contains a *text (code)* area, a *data* area and a BSS (blank common/zero initialised) area. Areas are the ‘atoms’ that Link deals with – it places them in the output image and resolves symbolic references between them.

Link's command line

Link's command line has the following format (optional items are enclosed by brackets '[' and ']'; braces '{' and '}' enclose items which may be repeated zero or more times):

```
link [<options>] <object-or-library-file> {<object-or-library-file>}
```

<object-or-library-file> is the name of an AOF or a.out format object or library file.

Under RISC OS, a filename ending in ".o" or ".O" is treated as the name of a directory in which the file identified by the previous word is to be sought (ie "clib.ansilib.o" refers to the file "clib.o.ansilib").

The difference between an object file and a library file is determined from the file's header. Sometimes it is necessary to differentiate between object files which are stored as libraries and true libraries. The /l qualifier appended to the filename indicates that it is to be treated as a library. If the /l qualifier is not specified an object type library is assumed; in this case all object modules in the library will be included. This only occurs with 'old style' libraries as generated by the Fortran compiler.

Object files are processed before libraries. Libraries are processed in the order they appear on the command line or in a via file. Libraries which depend on other libraries should be placed before the libraries they depend on. Mutually dependant libraries are not supported by the linker.

Filename arguments may be wildcarded according to the host system/shell's wildcarding rules; filename arguments to flag options (-output, -via, -edit, -overlay) may not be wildcarded.

Options

<options> is any compatible set of the following, case-insensitive flags:

- | | |
|--------------------------|---|
| -h[clp] | Print a screen of help text about the linker's command line format and terminate without doing anything, setting a good return code. |
| -o[output] <output-file> | Put the output in <output-file>. If this option is omitted, the name of the output file defaults as follows:

if generating a.out format then:
if running under Unix use "a.out"
otherwise "aout"

otherwise use the lower-case name of the output format (eg the default name of AIF output is "aif"). |

The following flags determine what kind of output file is generated. Except where noted, they are mutually exclusive. If none of the following is given, the default is -aif under RISC OS and -aout under Unix.

RISC OS oriented options

First, RISC OS oriented output formats:

<code>-ai[f]</code>	Generate an AIF image (the default under RISC OS). If there are undefined symbols, an error message is issued and no output is written.
<code>-r[elocatable]</code>	Generate a once off relocatable AIF image by adding a relocation table and self-relocation code to it. Such an image will run where it is loaded. This option sets the default output type to be AIF.
<code>-w[orkspace] <N></code>	Reserve <N> bytes of workspace for an image. If specified in conjunction with the <code>-relocatable</code> flag the linker produces a self-moving image which will move itself to the top of the application workspace at run time reserving <N> bytes of workspace above the image. (see <code>-base</code> , below, for a description of the allowed formats for N).
<code>-rm[f]</code>	Generate an RMF image including a relocation table and self relocation code to enable the image to be relocated at run time. This image may be relocated many times (eg. with the <code>*RMTidy</code> command)
<code>-m[odule]</code>	A synonym for <code>-rmf</code> .
<code>-ao[f]</code>	Generate partially linked AOF output suitable for inclusion in a subsequent link step.
<code>-bi[n]</code>	Generate a plain binary image with no file headers and contiguous read-only and read-write areas. This option is used with the <code>-base</code> flag to generate a plain memory image at a fixed base address.
<code>-db[ug]</code>	(Obsolescent). Generate output, in an obsolete image format, for use with the

Arthur low-level debugger Dbug. Do not confuse this option with `-debug` (described below).

`-ov[erlay] <overlay-file>`

Generate a RISC OS overlaid image as directed by commands in `<overlay-file>`.

Unix oriented options

Then, Unix-oriented output formats:

`-aou[t]`

Generate `a.out` (ZMAGIC) format output (the default under Unix). If there are undefined symbols, an error message is issued and no output is generated.

`-z[magic]`

A synonym for `-aout`.

`-om[agic]`

Generate partially linked `a.out` (OMAGIC) format suitable for inclusion in a subsequent link step.

Multiple format oriented options

The following options each apply to several image formats:

`-d[ebug]`

If the output format is `-aout`, then include the symbol table in the output for use by `dbx` or `adb`. If the output format is `-aif`, then include the symbol table in the output for use by the RISC OS debugger ASD (extended version). Do not confuse this option with the obsolescent `-dbug` (described above). With all other image formats, `-ddebug` is ignored.

`-en[try] <entry-point>`

Set the image's entry point to be `<entry-point>`. (Invalid with `-bin` and `-rmf`).

`-b[ase] <base address>`

Set the base address for the link operation to be `<base-address>`. (A sensible default is assumed for each image format.)

The numeric constant values given as arguments to the `-workspace`, `-base` and the `-entry` flags may be preceded by `'&'` or `'0x'` to denote a hexadecimal value; and may be followed by `'k'` (or `'K'`), denoting kilo- (&400), or `'m'` (or `'M'`), denoting Mega- (&100000). For example, `-base 80K`, `-base &50k`, `-base 1M`.

Miscellaneous options

The following options control miscellaneous features of the linker:

- | | |
|--|---|
| <code>-v[erbose]</code> | Print diagnostic information tracing the linker's progress. |
| <code>-c[ase]</code> | Ignore letter case when matching symbols. |
| <code>-v[ia] <via-file></code> | Take additional command lines from <code><via-file></code> . Each line of the command file is treated as a command line to link. Wildcarding is allowed for file names specified in the via file (in whatever form supported by the host system/shell). However, not all aspects of the host command line interface are available (eg RISC OS variable name substitution is not). |
| <code>-c[dit] <edit-file></code> | Perform link editing as instructed in <code><edit-file></code> . Refer to the section entitled <i>Link Editing</i> for further information. |

Other options are supported for backwards compatibility with the previous linker. These options are not documented and elicit warnings if used.

Linker pre-defined symbols

Link defines several useful symbolic names to which reference may be made. The pre-defined symbols occur in Base, Limit pairs. A Base value gives the address of the first byte in a region and the corresponding Limit value gives the address of the first byte beyond the end of the region. All pre-defined symbols begin "Image\$\$" and the space of all such names is reserved to Acorn. None of these symbols may be redefined. The pre-defined symbols are:

- | | |
|----------------------|--|
| Image\$\$RO\$\$Base | Address and limit of the Read-Only section of the image. |
| Image\$\$RO\$\$Limit | |

Image\$\$RW\$\$Base Address and limit of the Read-Write section
Image\$\$RW\$\$Limit of the image.

Image\$\$ZI\$\$Base Address and limit of the Zero-initialised data
Image\$\$ZI\$\$Limit section of the image.

If a section is absent, the Base and Limit values are equal but unpredictable.

Image\$\$RO\$\$Base includes any image header prepended by the linker.

Image\$\$RW\$\$Limit includes (at the end of the RW section) any zero-initialised data created at run-time.

Usually, language translators mark code as read-only and data as read-write so these three sections may be loosely thought of as code, data and zero-initialised data (in Unix terminology, text, data and bss).

Link editing

Link implements module composition via name matching. In such a world, name clashes can be a nuisance, even a disaster (consider two libraries, both obtained in object format and both containing a routine called 'plot'). In general, the constructors of libraries try to avoid 'name space pollution', but sometimes clashes are unavoidable. When clashes occur, link editing comes to the rescue.

Link editing has just 3 operations:

- 1 Composition of modules by matching named symbols.
- 2 Renaming of symbols (either on input or on output of a module).
- 3 Restriction of visibility (hiding) of symbols (either on input or on output of modules).

These operations can be shown to be universal and to permit any composition that can be specified, irrespective of naming clashes. Permitting renaming and hiding on both input and output of modules is, strictly, unnecessary, but is often convenient.

For example, old-style C programs are often polluted with variable and function names which don't need to be external. These are a potential source of name clashes. The Hide operation can remove the unnecessary external symbols from a partially linked object.

Appendix C - ARM Procedure Call Standard

Introduction

This appendix relates to the implementation of compiler code generators and language run-time library kernels for the Acorn RISC Machine (ARM).

The reader should be familiar with the ARM's instruction set [ARM], floating point instruction set [AFP] and assembler syntax [AASM] before attempting to use this information to implement a code generator. In order to write a run-time kernel for a language implementation, additional information specific to the relevant ARM operating system will be needed (some information is given in the sections describing the standard register bindings for this procedure-call standard).

The main topics covered herein are the procedure call and stack disciplines. These disciplines are followed by Acorn's C language implementation for the ARM and, eventually, will be followed by the Fortran and Pascal compilers too. Because C is the first-choice implementation language for RISC OS applications and the implementation language of Acorn's Unix product RISC iX, the utility of a new language implementation for the ARM will be related to its compatibility with Acorn's implementation of C.

At the end of this document are several examples of the usage of this standard, together with suggestions for generating effective code for the ARM.

Intent

The ARM Procedure Call Standard is a set of rules, designed:

- to facilitate calls between program fragments compiled from different source languages (eg to make subroutine libraries accessible to all compiled languages)
- to give compilers a chance to optimise procedure call, procedure entry and procedure exit (following the reduced instruction set philosophy of the ARM). These rules are used by the ARM's C compiler; implementors of other language translators are strongly encouraged to use them, too.

This standard defines the use of registers, the passing of arguments at an external procedure call, and the format of a data structure that can be used by stack backtracing programs to reconstruct a sequence of outstanding calls. It does so in terms of *abstract register names*. The binding of some register names to register numbers and the precise meaning of some aspects of the standard are somewhat dependent on the host operating system and are described in separate sections.

Formally, this standard only defines what happens when an *external procedure call* occurs. Language implementors may choose to use other mechanisms for internal calls and are not required to follow the register conventions described in this document except at the instant of an external call or return. However, other, system-specific invariants may have to be maintained if it is required, for example, to deliver reliably an asynchronous interrupt (eg a SIGINT) or give a stack backtrace upon an abort (eg when de-referencing an invalid pointer). More is said on this subject in later sections

Design Criteria

This procedure call standard was defined after a great deal of experimentation, measurement, and study of other architectures. It is believed to be the best compromise between the following important requirements:

- Procedure call must be extremely fast.
- The call sequence must be as compact as possible. (In typical compiled code, calls outnumber entries by a factor in the range 2-to-1 to 5-to-1.)
- Extensible stacks and multiple stacks must be accommodated. (The standard permits a stack to be extended in a non-contiguous manner, in *stack chunks*. The size of the stack does not have to be fixed when it is created, avoiding a fixed partition of the available data space between stack and heap. The same mechanism supports multiple stacks for multiple threads of control.)
- The standard should encourage the production of re-entrant programs, with writable data separated from code.
- The standard must support variation of the procedure call sequence, other than by conventional return from procedure (eg in support of C's longjmp, Pascal's goto-out-of-block, Modula-2+'s exceptions, Unix's signals, etc.) and tracing of the stack by debuggers and run-time error handlers. Enough is defined about the stack's structure to ensure that implementations of these are possible (within limits discussed later).

The procedure call standard

Register names

This section defines the standard.

The ARM has sixteen visible general registers and 8 floating-point registers.

Note: In interrupt modes some general registers are shadowed and not all floating-point operations are available, depending on how the floating-point operations are implemented.

This standard is written in terms of the *register names* defined in this section. The binding of certain register names (the *call frame registers*) to register numbers is discussed separately. We do this so that:

- Diverse needs can be more easily accommodated as can conflicting historical usage of register numbers, yet the underlying structure of the procedure call standard – on which compilers depend critically – remains fixed.
- Run-time support code written in assembly language can be made portable between different register bindings, if it obeys the rules given in the section entitled *Defined bindings*.

The register names and fixed bindings are given immediately below.

First, the four argument registers:

a1	RN	0	; argument 1/integer result
a2	RN	1	; argument 2
a3	RN	2	; argument 3
a4	RN	3	; argument 4

Then the six 'variable' registers:

v1	RN	4	; register variable
v2	RN	5	; register variable
v3	RN	6	; register variable
v4	RN	7	; register variable
v5	RN	8	; register variable
v6	RN	9	; register variable

General registers

Then the call-frame registers, the bindings of which vary (see the section on register bindings for details):

sl			; stack limit / stack chunk handle
fp			; frame pointer
ip			; temporary workspace, used in procedure entry
sp	RN	13	; lower end of current stack frame

Note that in the obsolete APCS-A register bindings described below, sp is bound to r12; in all other APCS bindings, sp is bound to r13.

Finally, lr and pc, which are determined by the ARM's hardware:

lr	RN	14	; link address on calls / temporary workspace
pc	RN	15	; program counter and processor status

Notes

Literal register names are given in lower case, eg v1, sp, lr. In the text that follows, symbolic values denoting "some register" or "some offset" are given in upper case, eg R, R+N.

References to "the stack" denoted by sp assume a stack that grows from high memory to low memory, with sp pointing at the top or front (ie lowest addressed word) of the stack.

At the instant of an external procedure call there shall be nothing of value to the caller stored below the current stack pointer, between sp and the (possibly implicit, possibly explicit) stack (chunk) limit. Whether there is a single stack chunk or multiple chunks, an explicit stack limit (in sl) or an implicit stack limit, is determined by the register bindings and conventions of the target operating system.

Here and in the text that follows, for any register R, the phrase "in R" refers to the contents of R; the phrase "at [R]" or "at [R, #N]" refers to the word pointed at by R or R+N, in line with ARM assembly language notation.

Floating point registers

The floating-point registers are divided into two sets, analogous to the subsets a1 - a4 and v1 - v6 of the general registers. Registers f0 - f3 need not be preserved by a called procedure; f0 is used as the floating-point result register. In certain restricted circumstances (noted below), f0 - f3 may be used to hold the first four floating-point arguments. Registers f4 - f7, the so called 'variable' registers, must be preserved by callees.

Data representation and argument passing

Register usage and argument passing

Control Arrival

The floating-point registers are:

f0	FN	0	; floating point result (or 1st FP argument)
f1	FN	1	; floating point scratch register (or 2nd FP arg)
f2	FN	2	; floating point scratch register (or 3rd FP arg)
f3	FN	3	; floating point scratch register (or 4th FP arg)
f4	FN	4	; floating point preserved register
f5	FN	5	; floating point preserved register
f6	FN	6	; floating point preserved register
f7	FN	7	; floating point preserved register

The ARM procedure call standard is defined in terms of $N (0)$ word-sized arguments being passed from the caller to the callee, and a single word or floating point result passed back by the callee. The standard does not describe the layout in store of records, arrays and so forth, used by ARM-targeted compilers for C, Pascal, Fortran-77, and so on. In other words, the mapping from language-level objects to APCS words is defined by each language's implementation, not by APCS, and, indeed, there is no formal reason why two implementations of, say, Pascal for the ARM should not use different mappings and, hence, not be cross-callable.

Obviously, it would be very unhelpful for a language implementor to stand by this formal position and implementors are strongly encouraged to adopt not just the letter of APCS but also the obviously natural mappings of source language objects into argument words. Strong hints are given about this in later sections which discuss (some) language specifics.

We consider the passing of $N (0)$ actual argument words to a procedure which expects to receive either exactly N argument words or a variable number $V (1)$ of argument words (it is assumed that there is at least one argument word which indicates in a language-implementation-dependent manner how many actual argument words there are, for example by using a format string argument, a count argument, or an argument-list terminator).

At the instant when control arrives at the target procedure, the following shall be true (for any M , if a statement is made about $\text{arg}M$, and $M > N$, the statement can be ignored):

- $\text{arg}1$ is in $\text{a}1$.
 $\text{arg}2$ is in $\text{a}2$.
 $\text{arg}3$ is in $\text{a}3$.
 $\text{arg}4$ is in $\text{a}4$.
for all $I \leq 5$, $\text{arg}I$ is at $[\text{sp}, \#4 * (I - 5)]$.
- fp contains 0 or points to a stack backtrace structure (as described in the next section).
- The values in sp , sl , fp are all multiples of 4.
- lr contains the $\text{pc} + \text{psw}$ value that should be restored into $\text{r}15$ on exit from the procedure. This is known as the *return link value* for this procedure call.
- pc contains the entry address of the target procedure.

Now, let us call the lower limit to which sp may point *in this stack chunk* "SP_LWM" (Stack-Pointer Low Water Mark). (Remember, it is unspecified whether there is one stack chunk or many, and whether SP_LWM is implicit, or explicitly derived from sl ; these are binding-specific details.) Then:

- space between sp and SP_LWM shall be (or shall be on demand) readable, writable memory which can be used by the called procedure as temporary workspace and overwritten with any values before the procedure returns.
- $\text{sp} \geq \text{SP_LWM} + 256$.

Note: this condition guarantees that a stack extension procedure, if used, shall have a reasonable amount – 256 bytes – of work space available to it, probably sufficient to call 2 or 3 procedure invocations further.

Control return

At the instant when the return link value for a procedure call is placed in the $\text{pc} + \text{psw}$, the following statements shall be true:

- fp , sp , sl , $\text{v}1 - \text{v}6$, and $\text{f}4 - \text{f}7$ shall contain the same values as they did at the instant of the call.
- If the procedure returns a word-sized result, R , which is not a floating point value, then R shall be in $\text{a}1$.

Notes

- If the procedure returns a floating point result, FPR, then FPR shall be in f0.

The definition of control return means that this is a "callee saves" standard.

The requirement to pass a variable number of arguments to a procedure (as in old-style C) precludes the passing of floating point arguments in floating point registers (as the ARM's fixed point registers are disjoint from its floating point registers). However, if a callee is defined to accept a fixed number K of arguments and its interface description declares it to accept exactly K arguments of matching types, then it is permissible to pass the first four floating point arguments in floating point registers f0 - f3.

Note: Acorn's C compiler for the ARM does not yet exploit this latitude.

The values of a2 - a4, ip, lr and f1 - f3 are not defined at the instant of return.

The Z, N, C and V flags are set from the corresponding bits in the return link value on procedure return. For procedures called using a BL instruction, these flag values will be preserved across the call.

Note: the flag values from lr at the instant of entry must be instated; it is not sufficient merely to preserve the flag values across the call. (Consider a procedure ProcA which has been "tail-call optimised" and does: CMPS a1, #0; MOVL T a2, #255; MOVGE a2, #0; B ProcB. If ProcB merely preserves the flags it sees on entry, rather than restoring those from lr, the wrong flags may be set when ProcB returns direct to ProcA's caller).

This standard does not define the values of fp, sp and sl at arbitrary moments during a procedure's execution, but only at the instants of (external) call and return. It should be noted that further standards and restrictions may apply under particular operating systems, to aid event handling or debugging. In general, you are strongly encouraged to preserve fp, sp and sl, at all times.

The minimum amount of stack defined to be available is not particularly great, and as a general rule a language implementation should not expect much more, unless the conventions of the target operating system indicate otherwise. For example, code generated by the RISC OS C compiler is able, if there is inadequate local workspace, to allocate more stack space from the C heap before continuing. Any language unable to do this may have its

interaction with C impaired. That `sl` contains a stack chunk handle is important in achieving this. (See the later discussion of RISC OS register bindings for further details).

The statements about `sp` and `SP_LWM` are designed to optimise the testing of the one against the other. For example, in the RISC OS user-mode binding of APCS, `sl` contains `SL_LWM+512`, allowing a procedure's entry sequence to include something like:

```
CMP    sp, sl
BLLT  |x$stack_overflow|
```

where `x$stack_overflow` is a part of the run-time system for the relevant language. If this test fails, and `x$stack_overflow` is not called, then:

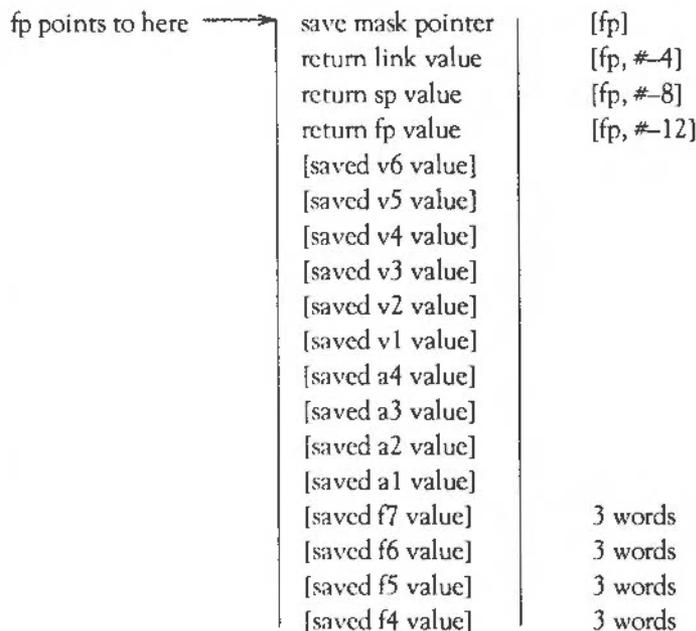
- there are at least 512 bytes free on the stack.

This procedure should only call other procedures when `sp` has been dropped by 256 bytes or less, guaranteeing that there is enough space for the called procedure's entry sequence (and, if needed, the stack extender) to work in.

If 256 bytes are not enough, the entry sequence has to drop `sp` before comparing with `sl` in order to force stack extension (see later sections on implementation specifics for details of how the RISC OS C compiler handles this problem).

Stack backtrace data structure

At the instant of an external procedure call, the value in *fp* is zero or it points to a data structure that gives information about the sequence of outstanding procedure calls. This structure is in the format shown below:



This picture shows between four and twenty-six words of store, with those words higher on the page being at higher addresses in memory. The values shown in square brackets are optional, and the presence of any does not imply the presence of any other. The floating point values are in extended format and occupy three words each.

At the instant of procedure call, all of the following statements about this structure shall be true:

- The *return fp value* is either 0 or contains a pointer to another stack backtrace data structure of the same form. Each of these corresponds to an active, outstanding procedure invocation. The statements listed here are also true of this next stack backtrace data structure and, indeed, hold true for each structure in the chain.

- The *save mask pointer* value, when bits 0, 1, 26, 27, 28, 29, 30, 31 have been cleared, points twelve bytes beyond a word known as the *return data save instruction*.
- The return data save instruction is a word that corresponds to an ARM instruction of the following form:

```

STMDB sp!, {[a1], [a2], [a3], [a4],
             [v1], [v2], [v3], [v4], [v5], [v6],
             fp, ip, lr, pc}

```

Note the square brackets in the above denote optional parts: thus, there are 12×1024 possible values for the return data save instruction, corresponding to the following bit patterns:

1110 1001 0010 1101 1101 10xx xxxx xxxx	APCS-R, APCS-U or
1110 1001 0010 1100 1100 11xx xxxx xxxx	APCS-A (obsolete)

The least-significant 10 bits represent argument and variable registers: if bit N is set, then register N will be transferred.

The optional parts [a1], [a2], [a3], [a4], [v1], [v2], [v3], [v4], [v5] and [v6] in this instruction correspond to those optional parts of the stack backtrace data structure that are present such that: for all M, if [vM] or [aM] is present then so is [saved vM value] or [saved aM value], and if [vM] or [aM] is absent then so is [saved vM value] or [saved aM value]. This is as if the stack backtrace data structure were formed by the execution of this instruction, following the loading of ip from sp (as is very probably the case).

- The sequence of up to four instructions following the return data save instruction determines whether saved floating point registers are present in the backtrace structure. The four optional instructions allowed in this sequence are:

```

STFE f7, [sp, #-12]! ; 11101101 01101101 01110001 00000011
STFE f6, [sp, #-12]! ; 11101101 01101101 01100001 00000011
STFE f5, [sp, #-12]! ; 11101101 01101101 01010001 00000011
STFE f4, [sp, #-12]! ; 11101101 01101101 01000001 00000011

```



Any or all of these instructions may be missing, and any deviation from this order or any other instruction terminates the sequence.

Note: an historical bug in the C compiler (now fixed) inserted a single arithmetic instruction between the return data save instruction and the first STFE. Some Acorn software allows for this.

Note that the bit patterns given are for APCS-R/APCS-U register bindings. In the obsolete APCS-A bindings, the bit indicated by the arrow is 0.

The optional instructions saving *f4*, *f5*, *f6* and *f7* correspond to those optional parts of the stack backtrace data structure that are present such that: for all *M*, if STFE *fM* is present then so is [saved *fM* value]; if STFE *fM* is absent then so is [saved *fM* value].

At the instant when procedure *A* calls procedure *B*, the stack backtrace data structure pointed at by *fp* contains exactly those elements [*v1*], [*v2*], [*v3*], [*v4*], [*v5*], [*v6*], [*f4*], [*f5*], [*f6*], [*f7*], *fp*, *sp* and *pc* which must be restored into the corresponding ARM registers in order to cause a correct exit from procedure *A*, albeit with an incorrect result.

Notes

The following example suggests what the entry and exit sequences for a procedure are likely to look like (though entry and exit are not defined in terms of these instruction sequences because that would be too restrictive; a good compiler can often do better than is suggested here):

```
entry  MOV    ip, sp
        STMDB sp!, {argRegs, workRegs, fp, ip, lr, pc}
        SUB   fp, ip, #4

exit   LDMDB  fp, {workRegs, fp, sp, pc}^
```

Many apparent idiosyncrasies in the standard may be explained by efforts to make the entry sequence work smoothly. This example above is neither complete (no stack limit checking) nor mandatory (making arguments contiguous for C, for instance, requires a slightly different entry sequence; and storing *argRegs* on the stack may be unnecessary).

The "workRegs" registers mentioned above correspond to as many of *v1* to *v6* that this procedure needs in order to work smoothly. At the instant when procedure *A* calls any other, those workspace registers not mentioned in *A*'s return data save instruction will contain the values they contained at the instant

A was entered. Additionally, the registers f4-f7 not mentioned in the floating point save sequence following the return data save instruction will also contain the values they contained at the instant A was entered.

This standard does not require anything of the values found in the optional parts [a1], [a2], [a3], [a4] of a stack backtrace data structure. They are likely, if present, to contain the saved arguments to this procedure call; but this is not required and should not be relied upon.

Defined bindings

APCS-R and APCS-U;
RISC OS and RISC iX

This section defines the bindings of the procedure call standard.

These bindings of the ARM procedure-call standard are used by:

- RISC OS applications running in ARM user-mode
- compiled code for RISC OS modules and handlers running in ARM SVC-mode
- RISC iX applications (which make no use of sl) running in ARM user mode
- RISC iX kernels running in ARM SVC mode.

The call-frame register bindings are:

sl	RN	10	; stack limit / stack chunk handle
			; unused by RISC iX applications
fp	RN	11	; frame pointer
ip	RN	12	; used as temporary workspace
sp	RN	13	; lower end of current stack frame

Although not formally required by this standard, it is considered good taste for compiled code to preserve the value of sl everywhere.

The invariants $sp > ip > fp$ have been preserved, in common with the obsolete APCS-A (described below), allowing symbolic assembly code (and compiler code-generators) written in terms of register names to be ported between APCS-R, APCS-U and APCS-A merely by re-labelling the call-frame registers provided:

- when call-frame registers appear in LDM, LDR, STM and STR instructions they are named symbolically, never by register numbers or register ranges;

Constraints on `sl` in
APCS-R

- no use is made of the ordering of the 4 call-frame registers (eg in order to load/save `fp` or `sp` from a full register save).

In SVC and IRQ modes (collectively called module mode) `SL_LWM` is implicit in `sp`: it is the next megabyte boundary below `sp`. Even though the SVC mode and IRQ mode stacks are not extensible, `sl` still points 512 bytes above a skeleton stack-chunk descriptor (stored just above the megabyte boundary). This is done for compatibility with use by applications running in ARM User mode and to facilitate module-mode stack-overflow detection. In other words:

$$sl = SL_LWM + 512.$$

When used in User mode, the stack is segmented and is extended on demand. (Acorn's language-independent run-time kernel allows language run-time systems to implement stack extension in a manner which is compatible with other Acorn languages). `sl` points 512 bytes above a full stack-chunk structure and, again:

$$sl = SL_LWM + 512.$$

Mode-dependent stack-overflow handling code in the language-independent run-time kernel faults an overflow in module mode and extends the stack in application mode. This allows library code, including the run-time kernel, to be shared between all applications and modules written in C.

In both modes, the value of `sl` must be valid immediately before each external call and each return from an external call.

Note: Deallocation of a stack chunk may be performed by intercepting returns from the procedure that caused it to be allocated. Tail-call optimisation complicates the relationship, so, in general, `sl` is required to be valid immediately before every return from external call.

Constraints on `sl` in
APCS-U

In this binding of the APCS the user-mode stack auto-extends on demand so `sl` is unused and there is no stack-limit checking.

In kernel mode, `sl` is reserved to Acorn.

The obsolete APCS-A binding

This obsolete binding of the procedure-call standard is used by Arthur applications running in ARM user-mode. The applicable call-frame register bindings are as follows:

sl	RN	13	; stack limit / stack chunk handle
fp	RN	10	; frame pointer
ip	RN	11	; used as temporary workspace
sp	RN	12	; lower end of current stack frame

Note: Use of r12 as *sp*, rather than the architecturally more natural r13, is historical and predates both Arthur and RISC OS.

In this binding of the APCS, the stack is segmented and is extended on demand. (Acorn's language-independent run-time kernel allows language run-time systems to implement stack extension in a manner which is compatible with other Acorn languages).

The stack limit register, *sl*, points 512 bytes above a stack-chunk descriptor, itself located at the low-address end of a stack chunk. In other words:

$$sl = SL_LWM + 512.$$

The value of *sl* must be valid immediately before each external call and each return from an external call.

Although not formally required by this standard, it is considered good taste for compiled code to preserve the value of *sl* everywhere.

Notes on APCS bindings

Invariants and APCS-M

In all future supported bindings of APCS *sp* shall be bound to r13.

In all supported bindings of APCS the invariant $sp > ip > fp$ shall hold.

This means that the only other possible binding of APCS is APCS-M:

sl	RN	12	; stack limit / stack chunk handle
fp	RN	10	; frame pointer
ip	RN	11	; used as temporary workspace
sp	RN	13	; lower end of current stack frame

Further restrictions in SVC and IRQ modes

There are some consequences of the ARM's architecture which, while not formally acknowledged by the ARM Procedure Call Standard, need to be understood by implementors of code intended to run in the ARM's SVC and IRQ modes.

An IRQ corrupts `r14_irq`, so IRQ mode code must run with IRQs off until `r14_irq` has been saved. Acorn's preferred solution to this problem is to enter and exit IRQ handlers written in high-level languages via hand-crafted "wrappers" which on entry save `r14_irq`, change mode to SVC, and enable IRQs and on exit restore the saved `r14_irq` (which restores IRQ mode and the IRQ-enable state). Thus the handlers themselves run in SVC mode, avoiding this problem in compiled code.

Both SWIs and aborts corrupt `r14_svc`. This means that care has to be taken when calling SWIs or causing aborts in SVC mode.

In high-level languages, SWIs are usually called out of line so it suffices to save and restore `r14` in the calling veneer around the SWI. If a compiler can generate in-line SWIs, then it should, of course, also save and restore `r14` in-line, around the SWI, in case the code has to run in SVC mode.

An abort in SVC mode may be symptomatic of a fatal error or it may be caused by page faulting in SVC mode. (Acorn expects SVC-mode code to be "correct", so these are the only options.) Page faulting can occur because an instruction needs to be fetched from a missing page (causing a prefetch abort) or because of an attempted data access to a missing page. The latter may occur even if the SVC-mode code is not itself paged (consider an unpagged kernel accessing a paged user-space).

A data abort is completely recoverable provided `r14` contains nothing of value at the instant of the abort. This can be ensured by:

- saving `R14` on entry to every procedure and restoring it on exit
- not using `R14` as a temporary register in any procedure
- avoiding page faults (stack faults) in procedure entry sequences.

A prefetch abort is harder to recover from and an aborting BL instruction cannot be recovered. So:

- special action has to be taken to protect page faulting procedure calls.

For Acorn C, r14 is saved in the 2nd or 3rd instruction of an entry sequence. Aligning all procedures at addresses which are 0 or 4 modulo 16 ensures that the critical part of the entry sequence cannot prefetch-abort. A compiler can do this by padding all code sections to a multiple of 16 bytes in length and being careful about the alignment of procedures within code sections.

Data-aborts early in procedure entry sequences can be avoided by using a software stack-limit check like that used in APCS-R.

Finally, the recommended way to protect BL instructions from prefetch-abort corruption is to precede each BL by a MOV ip, pc instruction. If the BL faults, the prefetch abort handler can safely overwrite r14 with ip before resuming execution at the target of the BL. If the prefetch abort is not caused by a BL then this action is harmless, as r14 has been corrupted anyway (and, by design, contained nothing of value at any instant a prefetch abort could occur).

Example procedure calls in C

Here is some sample assembly code as it might be produced by the C compiler:

```
; gggg is a function of 2 args that needs one register variable (v1)

gggg  MOV    ip, sp
      STMDB sp!, {a1, a2, v1, fp, ip, lr, pc}
      SUB   fp, ip, #4           ; points at saved PC
      CMPS  sp, sl
      BLLT |x$stack_overflow|   ; handler procedure
      ...
      MOV   v1, ...             ; use a register variable
      ...
      BL   ffff
      ...
      MOV   ..., v1            ; rely on its value after ffff()
```

Within the body of the procedure, arguments are used from registers, if possible; otherwise they must be addressed relative to fp. In the two argument case shown above, arg1 is at [fp,#-24] and arg2 is at [fp,#-20]. But as discussed below, arguments are sometimes stacked with positive offsets relative to fp.

Local variables are never addressed offset from fp; they always have positive offsets relative to sp. In code that changes sp this means that the offsets used may vary from place to place in the code. The reason for this is that it permits the procedure x\$stack_overflow to recover by setting sp (and sl) to some new

stack segment. As part of this mechanism, `x$stack_overflow` may alter memory offset from `fp` by negative amounts, eg `[fp, #-64]` and downwards, provided that it adjusts `sp` to provide workspace for the called routine.

If the function is going to use more than 256 bytes of stack it must go:

```
SUB    ip, sp, #<my stack size>
CMPS  ip, sl
BLLT  |x$stack_overflow_1|
```

instead of the two-instruction test shown above.

If a function expects no more than 4 arguments it can push all args onto the stack at the same time as saving its old `fp` and its return address (see the example above), and arguments are then saved contiguously in memory with `arg1` having the lowest address. A function that expects more than 4 arguments has code at its head as follows:

```
MOV    ip, sp
STMTD  sp!, {a1, a2, a3, a4}      ; put arg1-4 below stacked args
STMTD  sp!, {v1, v2, fp, ip, lr, pc} ; v1-v6 saved as necessary
SUB    fp, ip, #20               ; point at newly created call-frame
CMPS  sp, sl
BLLT  |x$stack_overflow|
...
...
LDMDB  fp, {v1, v2, fp, sp, pc}^ ; restore register vars & return
```

The entry sequence arranges that arguments (however many there are) lie in consecutive words of memory and that on return `sp` is always the lowest address on the stack that still contains useful data.

The time taken for a call, enter and return, with no arguments and no registers saved, is about 22 S-cycles.

Although not required by this standard, the values in `fp`, `sp` and `sl` are maintained while executing code produced by the C compiler. This makes it much easier to debug compiled code.

Multi-word results other than double precision reals in C programs are represented as an implicit first argument to the call, which points to where the caller would like the result placed. It is the first, rather than the last, so that it works with a C function that is not given enough arguments.

Procedure calls in other Acorn languages

Assembler

The procedure call standard is reasonably easy and natural for assembler programmers to use. The following rules should be followed:

Call-frame registers should always be referred to by explicitly by symbolic name, never by register number or implicitly as part of a register range.

The offsets of the call-frame registers within a register dump should not be wired into code. Always use a symbolic offset so that you can easily change the register bindings.

Fortran

The Acorn RISC OS Fortran-77 compiler violates the APCS in a number of ways that preclude inter-working with C, except via assembler veneers.

Pascal

The Acorn RISC OS ISO-Pascal compiler violates the APCS in a number of ways that preclude inter-working with C, except via assembler veneers.

Lisp, BCPL and BASIC

These languages have their own special requirements which make it inappropriate to use a procedure call of the form described here. Naturally, all are capable of making external calls of the given form, through a small amount of assembler "glue" code.

General

Note that there is no requirement specified by the standard concerning the production of re-entrant code, as this would place an intolerable strain on the conventional programming practices used in C and Fortran. The behaviour of a procedure in the face of multiple overlapping invocations is part of the specification of that procedure.

Various lessons

This document is not intended as a general guide to the writing of code generators, however it seems worthwhile to highlight various optimisations that appear particularly relevant to the ARM and to this standard.

The use of a callee-saving standard, instead of a caller-saving one, reduces the size of large code images by about ten percent (with compilers that do little or no interprocedural optimisation).

In order to make effective use of the APCS, compilers must compile code a procedure at a time. Line at a time compilation is insufficient.

The preservation of condition codes over a procedure call is often useful because any short sequence of instructions (including calls) that forms the body of a short IF statement can be executed without a branch instruction. For example:

```
if (a < 0) b = foo();
```

can compile into:

```
CMP    a, #0
BLLT  foo
MOVLT b, a1
```

In the case of a "leaf" or "fast" procedure, ie, one that calls no other procedures, much of the standard entry sequence can be omitted. In very small procedures, such as are frequently used in data abstraction modules, the cost of the procedure can be very small indeed. For instance, consider:

```
typedef struct {...; int a; ...} foo;

int get_a(foo* f) {return(f->a);}
```

The procedure geta can compile to just:

```
LDR    a1, [a1, #aOffset]
MOVS   pc, lr
```

This is also useful in procedures with a conditional as the top level statement, where one or other arm of the conditional is "fast" (ie calls no procedures). In this case there is no need to form a stack frame there. For example, using this, the C program:

```
int sum(int i)
{
    if (i <= 1)
        return(i);
    else
        return(i + sum(i-1));
}
```

could be compiled into:

```
sum    CMP    a1, #1      ; try fast case
       MOVSLD pc, lr     ; and if appropriate, handle quickly!

       ; else, form a stack frame and handle the rest as normal code.
       MOV    ip, sp
       STMDB  spl, {v1, fp, ip, lr, pc}
       CMP    sp, sl
       BLLT  overflow
       MOV    v1, a1      ; register to hold i
       SUB    a1, a1, #1  ; set up argument for call
       BL     sum         ; do the call
       ADD    a1, a1, v1  ; perform the addition
       LDMDB  fp, {v1, fp, sp, pc}^ ; and return
```

This is only worthwhile if the test can be compiled using only ip, and any spare of a1 - a4, as scratch registers. This technique can significantly speed up certain speed-critical routines, such as read and write character. At the present time, this optimisation is performed by the BCPL compiler but not by the C compiler.

Finally, it is often worth applying the "tail call" optimisation, especially to procedures which need to save no registers. For example, the code fragment:

```
extern void *malloc(size_t n)
{
    return primitive_alloc(NOTGCABLEBIT, BYTESTOWORDS(n));
}
```

is compiled by the C compiler into:

```
malloc ADD    a1, a1, #3      ; 1S
       MOV    a2, a1, #2     ; 1S
       MOV    a1, #10        ; 1S
       B     primitive_alloc ; 2N = 4S
```

This avoids saving and restoring the call-frame registers and minimises the cost of interface "sugaring" procedures. This saves 5 instructions and, on a 4/8 MHz ARM, reduces the cost of the malloc sugar from 24S to 7S.

References

If you need to find out more about ARM assembler and the ARM chip set, then refer to the following sources:

- ARM assembler is thoroughly covered in the manual supplied with the *ARM Assembler*, available from your Acorn supplier
- The ARM chip set is described in much greater detail in the *VL86C010 32-Bit RISC MPU and Peripheral User's Manual*, published by Prentice Hall.

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

Appendix D - ARM Object Format

Introduction

This document defines a file format called ARM Object Format, which is used by language processors for ARM-based systems. The AOF linker accepts input files in this format and generates output in the same format or in RISC OS Application Image Format. In the rest of this document, the term "object file" is used to denote a file in ARM Object Format and the term "linker" is used to denote the AOF linker.

Assumed terminology

Throughout this document the terms "byte", "half word", "word", and "string" are used to mean the following:

Byte	8 bits, considered unsigned unless otherwise stated, usually used to store flag bits or characters.
Half word	16 bits, or 2 bytes, usually unsigned. The least significant byte has the lowest address (DEC/Intel "byte sex", sometimes called "little endian"). The address of a half word (i.e. of its least significant byte) must be divisible by 2.
Word	32 bits, or 4 bytes, usually used to store a non-negative value. The least significant byte has the lowest address (DEC/Intel "byte sex", sometimes called "little endian"). The address of a word (i.e. of its least significant byte) must be divisible by 4.
String	A sequence of bytes terminated by a NUL (0x00) byte. The NUL is part of the string but is not counted in the string's length. Strings may be aligned on any byte boundary.

For emphasis: a word consists of 32 bits, 4-byte aligned; within a word, the least significant byte has the lowest address. This is DEC/Intel, or "little endian", *byte sex*, not IBM/Motorola *byte sex*.

Undefined Fields

Fields not explicitly defined by this document are implicitly reserved to Acorn. It is required that all such fields be zeroed. Acorn may ascribe meaning to such fields at any time, but will usually do so in a manner which gives no new meaning to zeroes.

Overall structure of an AOF file

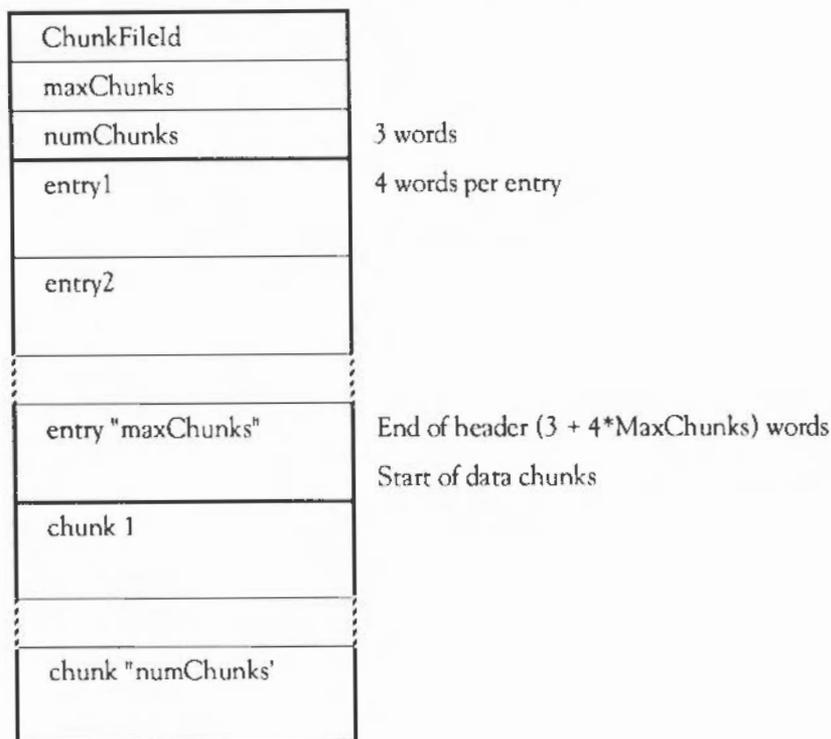
An object file contains a number of separate but related pieces of data. In order to simplify access to these data, and to provide for a degree of extensibility, the object file format is itself layered on another format called "Chunk File Format", which provides a simple and efficient means of accessing and updating distinct chunks of data within a single file. The object file format defines five chunks: "header", "areas", "identification", "symbol table", and "string table".

The minimum size of a piece of data in both formats is four bytes or one word. Each word is stored in a file in "little-endian" format; that is the least significant byte of the word is stored first.

Chunk file format

A chunk is accessed via a header at the start of the file. The header contains the number, size, location and identity of each chunk in the file. The size of the header may vary between different chunk files but is fixed for each file. Not all entries in a header need be used, thus limited expansion of the number of chunks is permitted without a wholesale copy. A chunk file can be copied without knowledge of the contents of the individual chunks.

Graphically, the layout of a chunk file is as follows:



ChunkFileId marks the file as a chunk file. Its value is C3CBC6C5 hex. The "maxChunks" field defines the number of the entries in the header, fixed when the file is created. The "numChunks" field defines how many chunks are currently used in the file, which can vary from 0 to "maxChunks". The value of "numChunks" is redundant as it can be found by scanning the entries.

Each entry in the header comprises four words in the following order:

- chunkId a two word field identifying what data the chunk contains file
- Offset a one word field defining the byte offset within the file of the chunk (which must be divisible by four); an entry of zero indicates that the corresponding chunk is unused
- size a one word field defining the exact byte size of the chunk (which need not be a multiple of four).

The "chunkId" field provides a conventional way of identifying what type of data a chunk contains. It is split into two parts. The first four characters (in the first word) contain a universally unique name allocated by a central authority (Acorn). The remaining four characters (in the second word) can be used to identify component chunks within this universal domain. In each part, the first character of the name is stored first in the file, and so on.

For AOF files, the first part of each chunk's name is "OBJ_"; the second components are defined in the next section.

Object file format

Each piece of an object file is stored in a separate, identifiable, chunk. AOF defines five chunks as follows:

Chunk	Chunk Name
Header	OBJ_HEAD
Areas	OBJ_AREA
Identification	OBJ_IDFN
Symbol Table	OBJ_SYMT
String Table	OBJ_STRT

Only the "header" and "areas" chunks must be present, but a typical object file will contain all five of the above chunks.

A feature of chunk file format is that chunks may appear in any order in the file. However, language processors which must also generate other object formats – such as Unix's a.out format – should use this flexibility cautiously.

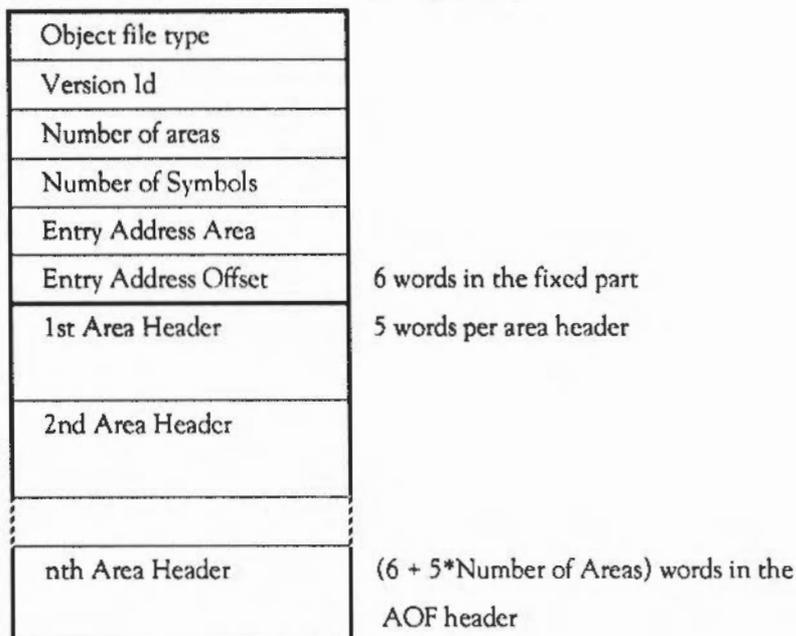
A language translator or other system utility may add additional chunks to an object file, for example a language-specific symbol table or language-specific debugging data, so it is conventional to allow space in the chunk header for additional chunks; space for eight chunks is conventional when the AOF file is produced a language processor which generates all five chunks described here.

The "header chunk" should not be confused with the chunk file's header.

Format of the AOF header chunk

The AOF header is logically in two parts, though these appear contiguously in the header chunk. The first part is of fixed size and describes the contents and nature of the object file. The second part is variable in length (specified in the fixed part) and is a sequence of "area" declarations defining the code and data areas within the OBJ_AREA chunk.

The AOF header chunk has the following format:



Object file type

C5E2D080 (hex) marks an object file as being in relocatable object format

Version ID

This word encodes the version of AOF to which the object file complies AOF 1.xx is denoted by 150 decimal; AOF 2.xx by 200 decimal.

Number of areas

The code and data of the object file is presented as a number of separate "areas", in the OBJ_AREA chunk, each with a name and some attributes (see below). Each area is declared in the (variable-length) part of the header which immediately follows the fixed part. The value of the "Number of Areas" field defines the number of areas in the file and consequently the number of area declarations which follow the fixed part of the header.

Number of symbols

If the object file contains a symbol table chunk "OBJ_SYMT", then this field defines the number of symbols in the symbol table.

Entry address area/ entry address offset

One of the areas in an object file may be designated as containing the start address for any program which is linked to include this file. If so, the entry address is specified as an <area-index, offset> pair, where "area-index" is in the range 1 to "Number of Areas", specifying the n'th area declared in the area declarations part of the header. The entry address is defined to be the base address of this area plus "offset".

A value of 0 for "area-index" signifies that no program entry address is defined by this AOF file.

Format of area headers

The area headers follow the fixed part of the AOF header. Each area header has the following form:

Area name			(offset into string table)
zeroes	AT	AL	
Area size			
Number of relocations			
Unused – must be zero			5 words in total

Area name

Each name in an object file is encoded as an offset into the string table, which is stored in the OBJ_STRT chunk. This allows the variable-length characteristics of names to be factored out from primary data formats. Each area within an object file must be given a name which is unique amongst all the areas in that object file.

AL

This byte must be set to 2; all other values are reserved to Acorn.

AT (Area attributes)

Each area has a set of attributes encoded in the AT byte. The least-significant bit of AT is numbered 0.

The linker orders areas in a generated image first by attributes, then by the (case-significant) lexicographic order of area names, then by position of the containing object module in the link-list. The position in the link-list of an object module loaded from a library is not predictable.

When ordered by attributes, Read-Only areas precede Read-Write areas which precede Debug areas; within Read-Only and Read-Write Areas, Code precedes Data which precedes Zero-Initialised data. Zero-Initialised data may not have the Read-Only attribute.

- Bit 0 This bit must be set to 0.
- Bit 1 If this bit is set, the area contains code, otherwise it contains data.
- Bit 2 Bit 2 specifies that the area is a common block definition.
- Bit 3 Bit 3 defines the area to be a (reference to a) common block and precludes the area having initialising data (see Bit 4, below). In effect, the setting of Bit 3 implies the setting of Bit 4.
- Common areas with the same name are overlaid on each other by the linker. The "Size" field of a common definition defines the size of a common block. All other references to this common block must specify a size which is smaller or equal to the definition size. In a link step there may be at most one area of the given name with bit 2 set. If none of these have bit 2 set, the actual size of the common area will be size of the largest common block reference (see also linker-defined symbols section).
- Bit 4 This bit specifies that the area has no initialising data in this object file and that the area contents are missing from the OBJ_AREA chunk. This bit is typically used to denote large uninitialised data areas. When an uninitialised area is included in an image, the linker either includes a read-write area of binary zeroes of appropriate size or maps a read-write area of appropriate size that will be zeroed at image start-up time. This attribute is incompatible with the read-only attribute (see "Bit 5", below).
- Note: Whether or not a zero-initialised area is re-zeroed if the image is re-entered is a property of the linker and the relevant image format. The definition of AOF neither requires nor precludes re-zeroing.

Relocation directives

If no relocation is specified, the value of a byte/halfword/word in the preceding area is exactly the value that will appear in the final image.

Bytes and halfwords may only be relocated by constant values of suitably small size. They may not be relocated by an area's base address.

A field may be subject to more than one relocation.

There are 2 types of relocation directive, termed here type-1 and type-2. Type-2 relocation directives occur only in AOF versions 150 and later.

Relocation can take two basic forms: "Additive" and "PCRelative".

Additive relocation specifies the modification of a byte/halfword/word, typically containing a data value (i.e. constant or address).

PCRelative relocation always specifies the modification of a branch (or branch with link) instruction and involves the generation of a program-counter-relative, signed, 24-bit word-displacement.

Additive relocation directives and type-2 PC-relative relocation directives have two variants: "Internal" and "Symbol".

Additive internal relocation involves adding the allocated base address of an area to the field to be relocated. With Type-1 internal relocation directives, the value by which a location is relocated is always the base of the area with which the relocation directive is associated (the SID field is ignored). In a type-2 relocation directive, the SID field specifies the index of the area relative to which relocation is to be performed. These relocation directives are analogous to the TEXT-, DATA- and BSS-relative relocation directives found in the a.out object format.

Symbol relocation involves adding the value of the symbol quoted.

A type-1 PCRelative relocation directive always references a symbol. The relocation offset added to any pre-existing in the instruction is the offset of the target symbol from the PC current at the instruction making the PCRelative reference. The linker takes into account the fact that the PC is eight bytes beyond that instruction.

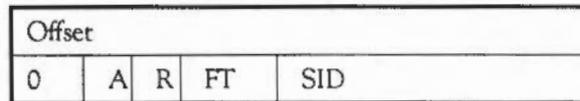
In a type-2 PC-relative relocation directive (only in AOF vsn 150 and later) the offset bits of the instruction are initialised to the offset from the base of the area of {the PC value current at the instruction making the reference} – thus the language translator, not the linker, compensates for the difference

between the address of the instruction and the PC value current at it. This variant is introduced in direct support of compilers that must also generate Unix's a.out format.

For a type-2 PC-relative symbol-type relocation directive, the offset added into the instruction making the PC-relative reference is the offset of the target symbol from the base of the area containing the instruction. For a type-2, PC-relative, internal relocation directive, the offset added into the instruction is the offset of the base of the area identified by the SID field from the base of the area containing the instruction.

The linker itself may generate type-2, PC-relative, internal relocation directives during the process of partially linking a set of object modules.

Diagrammatically:



Format of Type 1 relocation directives

Offset

Offset is the byte offset in the preceding area of the field to be relocated.

SID

If a symbol is involved in the relocation, this 16-bit field specifies the index within the symbol table (see below) of the symbol in question.

FT (Field Type)

This 2-bit field (bits 16 - 17) specifies the size of the field to be relocated:

- 00 byte
- 01 halfword
- 10 word
- 11 *illegal value*

R (relocation type)

This field (bit 18) has the following interpretation:

- 0 Additive relocation
- 1 PC-Relative relocation

A (Additive type)

In a type-1 relocation directive, this 1-bit field (bit 19) is only interpreted if bit 18 is a zero.

A=0 specifies "Internal" relocation, meaning that the base address of the area (with which this relocation directive is associated) is added into the field to be relocated. A=1 specifies "Symbol" relocation, meaning that the value of the given symbol is added to the field being relocated.

Bits 20 - 31

Bits 20-31 are reserved to Acorn and should be written as zeroes.

Format of Type 2
relocation directives

These are available from AOF 1.50 onwards.

Offset				
1000	A	R	FT	24-bit SID

The interpretation of *Offset*, *FT* and *SID* is exactly the same as for type-1 relocation directives except that *SID* is increased from 16 to 24 bits and has a different meaning – described below – if A=0).

The second word of a type-2 relocation directive contains 1 in its most significant bit; bits 28..30 must be written as 0, as shown.

The different interpretation of the *R* bit in type-2 directives has already been described in the section entitled *Relocation*.

If A=0 ('internal' relocation type) then *SID* is the index of the area, in the *OBJ_AREA* chunk, relative to which the value at *Offset* in the current area is to be relocated. Areas are indexed from 0.

Format of the symbol
table chunk

The "Number of Symbols" field in the header defines how many entries there are in the symbol table. Each symbol table entry has the following format:

Name	
	AT
Value	
Area name	

4 words per entry

Name

This value is an index into the string table (in chunk *OBJ.STRT*) and thus locates the character string representing the symbol.

AT

This is a 7 bit field specifying the attributes of a symbol as follows:

Bits 1 and 0

(10 means bit 1 set, bit 0 unset).

01 The symbol is defined in this object file and has scope limited to this object file (when resolving symbol references, the linker will only match this symbol to references from other areas within the same object file).

10 The symbol is a reference to a symbol defined in another area or another object file. If no defining instance of the symbol is found then the linker attempts to match the name of the symbol to the names of common blocks. If a match is found it is as if there were defined an identically-named symbol of global scope, having as value the base address of the common area.

11 The symbol is defined in this object file and has global scope (i.e. when attempting to resolve unresolved references, the linker will match this symbol to references from other object files).

00 Reserved to Acorn.

Bit 2

This attribute is only meaningful if the symbol is a defining occurrence (bit 0 set). It specifies that the symbol has an absolute value, for example, a constant. Otherwise its value is relative to the base address of the area defined by the "Area Name" field of the symbol table entry.

Bit 3

This bit is only meaningful if bit 0 is unset (that is, the symbol is an external reference). Bit 3 denotes that the reference is case-insensitive. When attempting to resolve such an external reference, the linker will ignore character case when performing the match.

Bit 4

This bit is only meaningful if the symbol is an external reference (bits 1,0 = 10). It denotes that the reference is "weak", that is that it is acceptable for the reference to remain unsatisfied and for any fields relocated via it to remain unrelocated.

Note: A weak reference still causes a library module satisfying that reference to be auto-loaded.

Bit 5	<p>This bit is only meaningful if the symbol is a defining, external occurrence (i.e. if bits 1,0 = 11). It denotes that the definition is "strong" and, in turn, this is only meaningful if there is a non-strong, external definition of the same symbol in another object file. In this scenario, all references to the symbol from outside of the file containing the strong definition are resolved to the strong definition. Within the file containing the strong definition, references to the symbol resolve to the non-strong definition.</p> <p>This attribute allows a kind of link-time indirection to be enforced. Usually, strong definitions will be absolute and will be used to implement an operating system's entry vector which must have the "forever binary" property.</p>
Bit 6	<p>This bit is only meaningful if bits 1,0 = 10. Bit 6 denotes that the symbol is a "common symbol" -- in effect, a reference to a common area with the symbol's name. The length of the common area is given by the symbol's value field (see below). The linker treats common symbols much as it treats areas having the "common reference" bit set -- all symbols with the same name are assigned the same base address and the length allocated is the maximum of all specified lengths.</p> <p>If the name of a common symbol matches the name of a common area then these are 'merged' and symbol identifies the base of the area.</p> <p>All common symbols for which there is no matching common area (reference or definition) are collected into an anonymous linker pseudo-area.</p>
Value	<p>This field is only meaningful if the symbol is a defining occurrence (i.e. bit 0 of AT set) or a common symbol (i.e. bit 6 of AT set). If the symbol is absolute (bit 2 of AT set), this field contains the value of the symbol. Otherwise, it is interpreted as an offset from the base address of the area defined by "Area Name", which must be an area defined in this object file.</p>
Area name	<p>This field is only meaningful if the symbol is not absolute (i.e. if bit 2 of AT is unset) and the symbol is a defining occurrence (i.e. bit 0 of AT is set). In this case it gives the index into the string table of the character string name of the (logical) area relative to which the symbol is defined.</p>
The string table chunk (OBJ_STRT)	<p>The string table chunk contains all the print names referred to within the "areas" and "symbol table" chunks. The separation is made to factor out the variable length characteristic of print names. A print name is stored in the</p>

string table as a sequence of ISO8859 non-control characters terminated by a NUL (0) byte and is identified by an offset from the table's beginning. The first 4 bytes of the string table contain its length (including the length word – so no valid offset into the table is less than 4 and no table has length less than 4). The length stored at the start of the string table itself is identically the length stored in the OBJ_STRT chunk header.

The identification chunk (OBJ_IDFN)

This chunk should contain a character string (excluding non-whitespace codes in the ranges [0..31] and 128+[0..31]), terminated by a NUL (0) byte, giving information about the name and version of the language translator which generated the object file.

Linker defined symbols

Though not part of the definition of AOF, the definitions of symbols which the AOF linker defines during the generation of an image file are collected here. These may be referenced from AOF object files, but must not be redefined.

Linker pre-defined symbols

The pre-defined symbols occur in Base, Limit pairs. A Base value gives the address of the first byte in a region and the corresponding Limit value gives the address of the first byte beyond the end of the region. All pre-defined symbols begin "Image\$\$" and the space of all such names is reserved to Acorn.

None of these symbols may be redefined. The pre-defined symbols are:

Image\$\$RO\$\$Base	Address and limit of the Read-Only section of the image.
Image\$\$RO\$\$Limit	
Image\$\$RW\$\$Base	Address and limit of the Read-Write section of the image.
Image\$\$RW\$\$Limit	
Image\$\$ZI\$\$Base	Address and limit of the Zero-initialised data section of the image (created from areas having bit 4 of their area attributes set and from "common symbols" which match no area name).
Image\$\$ZI\$\$Limit	

If a section is absent, the Base and Limit values are equal but unpredictable.

Image\$\$RO\$\$Base includes any image header prepended by the linker.

Image\$\$RW\$\$Limit includes (at the end of the RW section) any zero-initialised data created at run-time.

The Image\$\$xx\$\${Base,Limit} values are intended to be used by language run-time systems. Other values which are needed by a debugger or by part of the pre-run-time code associated with a particular image format are deposited into the relevant image header by the Linker.

Common area symbols

For each common area, the linker defines a global symbol having the same name as the area, except where this would clash with the name of an existing global symbol definition (thus a symbol reference may match a common area).

Obsolescent and obsolete features

The following sub-sections describe features that were part of revision 1.xx of AOF and/or that were supported by the 59x releases of the AOF linker, which are no longer supported. In each case, a brief rationale for the change is given.

Object file type

AOF used to define three image types as well as a relocatable object file type. Image types 2 and 3 were never used under Arthur/RISC OS and are now obsolete. Image type 1 is used only by the obsolescent Dbug (new releases of a debugger having Dbug's functionality will use Application Image Format).

AOF Image type 1	C5E2D081 hex	(obsolescent)
AOF Image type 2	C5E2D083 hex	(obsolete)
AOF Image type 3	C5E2D087 hex	(obsolete)

AL (Area alignment)

AOF used to allow the alignment of an area to be any specified power of 2 between 2 and 16. By convention, relocatable object code areas always used minimal alignment (AL=2) and only the obsolete image formats, types 2 and 3, specified values other than 2. From now on, all values other than 2 are reserved to Acorn.

AT (Area attributes)

Two attributes have been withdrawn: the Absolute attribute (bit 0 of AT) and the Position Independent attribute (bit 6 of AT).

The Absolute attribute was not supported by the RISCOS linker and therefore had no utility. The next linker release will, in any case, allow the effect of the absolute attribute to be simulated.

The Position Independent bit used to specify that a code area was position independent, meaning that its base address could change at run-time without any change being required to its contents. Such an area could only contain internal, PC-relative relocations and must make all external references through registers. Thus only code and "pure data" (containing no address values) could be position-independent.

Few language processors generated the PI bit which was only significant to the generation of the obsolete image types 2 and 3 (in which it affected AREA placement). Accordingly, its definition has been withdrawn.

Fragmented areas

The concept of fragmented areas was introduced in release 0.04 of AOF, tentatively in support of Fortran compilers. To the best of our knowledge, fragmented areas were never used. (Two warnings against use were given with the original definition on the grounds of: structural incompatibility with Unix's a.out format; and likely inefficient handling by the linker. And use was hedged around with curious restrictions). Accordingly, the definition of fragmented areas is withdrawn.

Appendix E - File formats

Introduction

The file formats described in this appendix are those generated by RISC OS itself and various applications. Each is shown as a chart giving the size and description of each element. The elements are sequential and the sizes are in bytes.

This appendix contains information about the following file formats:

- Sprite files
- Template files
- Draw files
- Font files, including IntMetrics and font files
- Music files

Sprite files

A sprite file is saved in the same format as a sprite area is in memory, except that the first word of the sprite is not saved.

For a full description about sprite formats, refer to the section *Technical Details* in the chapter entitled *Sprites*.

Template files

The following section describes the Wimp template file format:

	Offset	Size	Meaning
Header:			
	0	4	file offset of font data (-1 ==> none)
	4	4	0
	8	4	0
	12	4	0
	16		

Index entries:

n+0	4	file offset of data for this entry
n+4	4	size of data for this entry
n+8	4	entry type (1 = window)
n+12	12	identifier (terminated by ASCII 13)
n+24		

Terminator:

(next)	4	0
--------	---	---

Data:

d+0	88	window definition (as in Wimp_CreateWindow)
d+88	ni*32	icon definitions
(next)		indirected icon data

Any pointers to indirected icon data are the file offsets. Any references to anti-aliased fonts use internal handles.

Font data:

f+0	4	x-point-size * 16
f+4	4	y-point-size * 16
f+8	40	font name (terminated by <cr>
f+48		

The first font entry is that referred to by internal handle 1, the second font entry is that referred to by internal handle 2, etc.

Draw files

The Draw file format provides an object-oriented description of a graphic image. It represents an object in its editable form, unlike a page-description language such as PostScript which simply describes an image.

Programmers wishing to define their own object types should use the same approach as for the allocation of SWI numbers.

Coordinates

All coordinates within a Draw file are signed 32-bit integers that give absolute positions on a large image plane. The units are $1/(180*256)$ inches, or $1/640$ of a printer's point. When plotting on a standard RISC OS screen, an assumption is made that one OS-unit on the screen is $1/180$ of an inch. This gives an image reaching over half a mile in each direction from the origin. The actual image size (eg. the page format) is not defined by the file, though the maximum extent of the objects defined is quite easy to calculate. Positive-x is to the right, Positive-y is up. The printed page conventionally has the origin at its bottom left hand corner. When rendering the image on a raster device, the origin is at the bottom left hand corner of a device pixel.

Colours

Colours are specified in Draw files as absolute RGB values in a 32-bit word. The format is:

Byte	Description
0	reserved. Must be zero
1	unsigned red value
2	unsigned green value
3	unsigned blue value

For colour values, 0 means none of that colour and 255 means fully saturated in that colour.

You must always write byte 0 (the reserved one) as a zero, but don't assume that it always will be 0 when reading.

The bytes in a word of an Draw file are in little-endian order, eg the least significant byte appears first in the file.

The special value &FFFFFFF is used in the filling of areas and outlines to mean "transparent".

File headers

The file consists of a header, followed by a sequence of objects.

The file header is of the following form.

Size	Description
4	"Draw"
4	Major format version stamp – currently 201 (decimal)
4	Minor format version stamp – currently 0
12	Identity of the program that produced this file. Typically 8 ASCII chars, padded with spaces.
16	Bounding box: low x, low y, high x, high y

When rendering a Draw file, check the major version number. If this is greater than the latest version you recognise then refuse to render the file (eg. generate an error message for the user), as an incompatible change in the format has occurred.

The entire file is rendered by rendering the objects one by one, as they appear in the file.

The bounding box indicates the intended image size for this drawing.

A Draw file containing a file header but no objects is legal; however, the bounding box is undefined. In particular the 'x0' value may be greater than the 'x1' value (and similarly for the y values).

Object header

Each object, with the exception of the font table object, consists of an object header, followed by a variable amount of data depending on the object type. The object header is of the following form:

Size	Description
4	Object type field
4	Object size: number of bytes in the object. Always a multiple of 4
16	Object bounding box: low x, low y, high x, high y

The bounding box describes the maximum extent of the rendition of the object: the object cannot affect the appearance of the display outside this rectangle. The upper coordinates are an outer bound, in that the device pixel

at (x-low, y-low) may be affected by the object, but the one at (x-high, y-high) may not be. The rendition procedure may use clipping on these rectangles to abandon obviously invisible objects.

Objects with no direct effect on the rendition of the file have no bounding box: these will be identified explicitly in the object descriptions that follow. If an unidentified object type field is encountered when rendering a file, ignore the object and continue.

The rest of the data for an object depends on the object type.

Font table object

Object type number 0

This is followed by a sequence of font number definitions

Size	Description
1	Font number (non-zero)
n	n character textual font name, null terminated
0 - 3	The list is terminated by up to 3 zero characters, to pad to word boundary

This object type is somewhat special in that only one instance of it ever appears in a Draw file. It has no direct effect on the appearance of the image, but maps font numbers (used in text objects) to textual names of fonts. It must precede all Text objects. A Font Table object has no bounding box in its object header. Comparison of font names is case-insensitive.

Text object

Object type number 1

Size	Description
4	Text colour
4	Text background colour hint
4	Text style
4	X unsigned nominal size of the font (in 1/640 point)
4	Y unsigned nominal size of the font (in 1/640 point)
8	X,Y coordinates of the start of the text base line
n	n characters in the string, null terminated
0 - 3	Up to 3 zero characters, to pad to word boundary

The character string consists of printing ANSI characters with codes within 32 - 126 or 128 - 255. This need not be checked during rendering, but other codes may produce undefined or system-dependent results.

The style word consists of the following:

bits 0 - 7	one byte font number
bits 8 - 31	reserved (must be zero)

Italic, bold etc. variants are assumed to be distinct fonts.

The font number is related to the textual name of a font by a preceding Font Table object within the file (see above). The exception to this is font number 0, which is a system font that is sure to be present. When rendering a draw file, if a font is not recognised, the system font should be used instead. The system font is monospaced, with the gap between letters equal to the x nominal size of the font.

The background colour hint can be used by font rendition code when performing anti-aliasing. It is referred to as a hint because it has no effect on the rendition of the object on a "perfect" printer, nevertheless for screen rendition it can improve the appearance of text on coloured backgrounds. The font rendition code can assume that the text appears on a background that matches the background colour hint.

Path object

Object type number 2

Size	Description
4	Fill colour (-1 means do not fill)
4	Outline colour (-1 means no outline)
4	Outline width (unsigned)
4	Path style description
	Optional dash pattern definition
	Sequence of path components

An outline width of 0 means draw the thinnest possible outline that the device can represent. A path component consists of:

4	1-word tag identifier
n	Sequence of 2-word (x,y) coordinate pairs

Each tag identifier word consists of:

Bit(s)	Description
0 - 7	Tag identifier byte
8 - 31	Reserved, must be zero

Possible tag identifier byte values are:

0	end of path: no arguments
2	move to absolute position: followed by x,y pair
5	close current sub-path
8	draw to absolute position: followed by x,y pair
6	Bezier curve through two control points, to absolute position followed by three x,y pairs

The tag values match the arguments required by the Draw module. This block of memory can be passed directly to the Draw module for rendition, see the relevant documentation for precise rules concerning the appearance of paths. See also manuals on PostScript [Reference: PostScript Language Reference Manual, Addison-Wesley]).

The possible set of legal path components in a path object is described as follows. A path consists of a sequence of (at least one) subpaths, followed by an "end of path" path component. A subpath consists of a "move to" path component, followed by a sequence of (at least one) "draw to" and "Bezier to" path components, followed (optionally) by a "close sub-path" path component.

The path style description word is as follows:

Bit(s)	Description
0 - 1	join style: 0 = mitred joins 1 = round joins 2 = bevelled joins
2 - 3	end cap style: 0 = butt caps 1 = round caps 2 = projecting square caps 3 = triangular caps
4 - 5	start cap style (same possible values)
6	winding rule: 0 = non-zero

7	1 = even-odd dash pattern: 0 = dash pattern missing 1 = dash pattern present
8 - 15	reserved and must be zero
16 - 23	triangle cap width: a value within 0 - 255, measured in sixteenths of the line width.
24 - 31	triangle cap length: a value within 0 - 255, measured in sixteenths of the line width.

The mitre cut-off value is the PostScript default (eg. 10). If the dash pattern is present then it is in the following format:

Size	Description
4	Offset distance into the dash pattern to start
4	Number of elements in the dash pattern
4	For each element of the dash pattern, Length of the dash pattern element
	The dash pattern is reused cyclically along the length of the path, with the first element being filled, the next a gap, and so on.

Sprite object

Object type number 5

This is followed by a sprite block. See the chapter entitled *Sprites* for details.

This contains a pixmap image. The image is scaled to entirely fill the bounding box.

If the sprite has a palette then this gives absolute values for the various possible pixels. If the sprite has no palette then colours are defined locally. Within RISC OS the available "Wimp colours" are used – see the chapters entitled *Sprites* and *The Window Manager* for further details.

Group object

Object type number 6

Size	Description
12	Group object name, padded with spaces

This is followed by a sequence of other objects.

The objects contained within the group will have rectangles contained entirely within the rectangle of the group. Nested grouped objects are allowed.

The object name has no effect on the rendition of the object. It should consist entirely of printing characters. It may have meaning to specific editors or programs. It should be preserved when copying objects. If no name is intended, twelve space characters should be used.

Tagged object

Object type number 7

Size	Description
------	-------------

4	Tag identifier
---	----------------

This is followed by an object and optional word-aligned data

To render a Tagged object, simply render the enclosed object. The identifier and additional data have no effect on the rendition of the object. This allows specific programs to attach meaning to certain objects, while keeping the image renderable.

Programmers wishing to define their own object tags should use the same approach as for the allocation of SWI numbers.

Text area object

Object type number 9

Size	Description
------	-------------

1	Object type number: 9
---	-----------------------

1 or more text column objects (object type 10, no data – see below):

4	Zero, to mark the end of the text columns
---	---

4	Reserved, which must be zero
---	------------------------------

4	Reserved, which must be zero
---	------------------------------

4	Initial text foreground colour
---	--------------------------------

4	Initial text background colour hint
---	-------------------------------------

The body of the text column: ASCII characters, terminated by a null character.

0 - 3	Null characters to align to a word boundary
-------	---

A text area contains a number of text columns. The text area has a body of text associated with it, which is partitioned between the columns. If there is just one text column object then its bounding box must be exactly coincident with that of the text area.

The body of the text is paginated in the columns. Effects such as font settings and colour changes are controlled by escape sequences within the body of the text. All escape sequences start with a backslash character (\); the escape code is case sensitive, though any arguments it has are not.

Arguments of variable length are terminated by a `'/` or `<newline>`. Arguments of fixed length are terminated by an optional `'/`.

Values with range limits mean that if a value falls outside the range, then the value is truncated to this limit.

Escape Sequence	Description
-----------------	-------------

- `\! <version><newline or />`
Must appear at the start of the text, and only there. `<version>` must be 1.
- `\A<code><optional />`
Set alignment. `<code>` is one of L (left = default), R (right), C (centre), D (double). A change in alignment forces a line break.
- `\B<R><spaces><G><spaces><newline or />`
Set text background colour hint to the given RGB intensity (or the best available approximation). Each value is limited to 0 - 255.
- `\C<R><spaces><G><spaces><newline or />`
Set text foreground colour to the given RGB intensity (or the best available approximation). Each value is limited to 0 - 255.
- `\D<number><newline or />`
Indicates that the text area is to contain `<number>` columns. Must appear before any printing text.

- `\F<digit*><name><spaces><size>[<spaces><width>]<newline or />`
 Defines a font reference number. `<name>` is the name of the font, and `<size>` its height. `<width>` may be omitted, in which case the font width and height are the same. Font reference numbers may be reassigned. `<digit*>` is one or two digits. `<size>` and `<width>` are in points.
- `\<digit*><optional />`
 Selects a font, using the font reference number
- `\L<leading><newline or />`
 Define the leading in points from the end of the (output) line in which the `\L` appears. ie the vertical separation between the bases of characters on separate lines. Default, 10 points.
- `\M<leftmargin><spaces><rightmargin><newline or />`
 Defines margins that will be left on either side of the text, from the start of the output line in which the sequence appears. The margins are given in points, and are limited to values > 0 . If the sum of the margins is greater than the width of the column, the effects are undefined. Both values default to 1 point.
- `\P<leading><newline or />`
 Define the paragraph leading in points, ie the vertical separation between the end of one paragraph and the beginning of a new paragraph. Default, 10 points.
- `\U<position><spaces><thickness><newline or />`
 Switch on underlining, at `<position>` units relative to the character base, and of `<thickness>` units, where a unit is $1/256$ of the current font size. `<position>` is limited to $-128\dots+127$, and `<thickness>` to $0\dots255$. To turn the underlining off, either give a thickness of 0, or use the command `'\U.'`
- `\V[-]<digit><optional />`
 Vertical move by the specified number of points.

- \- Soft hyphen: if a line cannot be split at a space, a hyphen may be inserted at this point instead; otherwise, it has no printing effect.
- \<newline> Force line break.
- \\ appears as \ on the screen
- \;<text><newline> Comment: ignored.

Other escape sequences are flagged as errors during verification.

Lines within a paragraph are split either at a space, or at a soft hyphen, or (if a single word is longer than a line) at any character.

A few other characters have a special interpretation:

- Control characters are ignored, except for tab, which is replaced by a space.
- Newlines (that are not part of an escape sequence) are interpreted as follows:
 - occurring before any printing text. A paragraph leading is inserted for each newline.
 - in the body of the text. A single newline is replaced by a space, except when it is already followed or preceded by a space or tab. A sequence of n newlines inserts a space of (n-1) times the paragraph leading, except that paragraph leading at the top of a new text column is ignored.

Lines which protrude beyond the limits of the box vertically, eg. because the leading is too small, are not displayed; however, any font changes, colour changes, etc. in the text are applied. Characters should not protrude horizontally beyond the limits of the text column, eg. due to italic overhang for this font being greater than the margin setting.

Restrictions

If a chunk of text contains more than 16 colour change sequences, only the last 16 will be rendered correctly. This is a consequence of the size of the colour cache used within the font manager. A chunk in this case means a block of text up to anything that forces a line break, eg. the end of a paragraph or a change on the alignment.

Text column object

Object type number 10

No further data, ie just an object header.

A text column object may only occur within a text area object. Its use is described in the section on text area objects.

Font files

Fonts are described in:

- IntMetrics
- x90y45 files (old style 4-bpp bitmaps)
- New font file formats

IntMetrics

Size	Description
40	Name of font, padded with Return characters
4	16
4	16
1	n = number of defined characters
3	reserved – currently 0
256	character mapping (ie. indices into following arrays). For example, if the 40th byte in this block is 4, then the fourth entry in each of the following arrays refers to that character. A zero entry means that character is not defined in this font.
2n	x0
2n	y0 bounding box of character (in 1/1000ths em)
2n	x1 coordinates are relative to the 'origin point'
2n	y1
2n	x-offset after printing this character
2n	y-offset after printing this character

The bounding boxes and offsets are given as 16-bit signed numbers, with the low byte first.

x90y45 font files

Each font file starts with a series of 4-word index entries, corresponding to the sizes defined:

Size	Description
1	point size (not multiplied by 16)
1	bits per pixel (4)
1	pixels per inch (x-direction)
1	pixels per inch (y-direction)
4	reserved – currently 0
4	offset of font data in file
4	size of font data

The list is terminated by:

1 0

Font data

Font data is limited to 64K per block. Each block starts word-aligned relative to the start of the file:

Size	Description
4	x-size in 1/16ths point * x pixels per inch
4	y-size in 1/16ths point * y pixels per inch
4	pixels per inch in the x-direction
4	pixels per inch in the y-direction
1	x0 maximum bounding box for any character
1	y0 bottom-left is inclusive
1	x1 top-right is exclusive
1	y1 all coordinates are in pixels
512	2-byte offsets from table start of character data. A zero value means the character is not defined. These are low/high byte pairs (ie little-endian)

Character data

Size	Description
1	x0 bounding box
1	y0
1	x1 - x0 = X
1	y1 - y0 = Y
X*Y/2	4-bits per pixel (bpp), consecutive rows bottom to top. Not aligned until the end.
0 - 3.5	Alignment

New font file formats

The new font file formats includes definitions for the following types of font files:

- f9999x9999 (new style 4-bpp anti-aliased fonts)
- b9999x9999 (1-bpp bitmaps)
- outlines (outline font format, for all sizes)

'9999' = pixel size (ie point size * 16 * dpi / 72) zero-suppressed decimal number.

File header

If the length of an outlines file is less than 256 bytes, then contents are the name of another font whose glyphs are to be used instead (with this fonts metrics).

If the length of a x90y45 file is less than 256 bytes then contents are the name of the f9999x9999 file to use as master bit maps.

The file header is of the following form :

Size	Description
4	"FONT" – identification word
1	Bits per pixel: 0 = outlines 1 = 1 bpp 4 = 4 bpp
1	Version number of file format 4: no "don't draw skeleton lines unless smaller than this" byte present 5: byte at [table+512] = max pixel size for skeleton lines (0 => always do it) 6: byte at [chunk+indexsize] = dependency mask (see below)
2	if bpp = 0: design size of font if bpp > 0: flags: bit 0 set – horizontal subpixel placement bit 1 set – vertical subpixel placement
2	x0 – font bounding box (16-bit signed)
2	y0 – units are pixels or design units
2	x1 – x0 : x0,y0 inclusive, x1,y1 exclusive
2	y1 – y0
4	file offset of 0...31 chunk (word-aligned)
4	file offset of 32...63 chunk
...	
4	file offset of 224...255 chunk
4	file offset of end (ie. size of file) if offset(n+1)=offset(n), then chunk n is null.

Table start:

2 n = size of table/scaffold data

Bitmaps: (n=10 normally – other values are reserved)

2 x-size (1/16th point)

2 x-res (dpi)

2 y-size (1/16th point)

2 y-res (dpi)

Outlines:

510 offsets of scaffold data from table start

0 => no scaffold data for char

1 Skeleton threshold pixel size

(if file format version >= 5)

When rastering the outlines, skeleton lines will only be drawn if either the x-or y- pixel size is less than this value (except if value=0, which means 'always draw skeleton lines').

? ... sets of scaffold data (see below)

Table end:

? description of contents of file:

, 0, "Outlines", 0

"999x999 point at 999x999 dpi", 0

... word-aligned chunks follow

Scaffold data:

1 char code of 'base' scaffold entry (0 ==> none)

1 bit n set ==> x-scaffold line n defined in base char

1 bit n set ==> y-scaffold line n defined in base char

1 bit n set ==> x-scaffold line n defined locally

1 bit n set ==> y-scaffold line n defined locally

... local scaffold lines follow

Scaffold lines:

- 2 bits 0 - 11 = coordinate (signed)
bits 12 - 14 = scaffold link index (0 => none)
bit 15 set => 'linear' scaffold link
- 1 width (254 ==> L-rangent, 255 ==> R-rangent)

Chunk data:

- 4 * 32 offset within chunk to character
0 => character is not defined
 - * 4 for vertical placement
 - * 4 for horizontal placementCharacter index is more tightly bound than vertical placement which is more tightly bound than horizontal placement.
- 1 Dependency byte (if outline file, version >= 6).
Bit n set => chunk n must be loaded in order to rasterise this chunk. This is required for composite characters which include characters from other chunks (see below).

Note: All character definitions must follow the index in the order in which they are specified in the index. This is to allow the font editor to easily determine the size of each character.

... word-aligned character data follows

Char data:

- 1 flags:
 - bit 0 set => coords are 12-bit, else 8-bit
 - bit 1 set => data is 1-bpp, else 4-bpp
 - bit 2 set => initial pixel is black, else white
 - bit 3 set => data is outline, else bitmap
 - bits 4 - 7 = 'p' value for char (0 ==> not encoded)
- 2/3 x0, y0 sign-extended 8- or 12- bit coordinates
- 2/3 xs, ys width, height (bbox = x0,y0,x0+xs,y0+ys)
- n data: (depends on type of file)
 - 1-bpp uncrunched: rows from bottom to top
 - 4-bpp uncrunched: rows from bottom to top

Outline char format

1-bpp crunched: list of (packed) run-lengths
outlines: list of move/line/curve segments
word-aligned at the end of the character data

Here the 'pixel bounding box' is actually the bounding box of the outline in terms of the design size of the font (in the file header). The data following the bounding box consists of a series of move/line/curve segments followed by a terminator and an optional extra set of line segments followed by another terminator. When constructing the bitmap from the outlines, the font manager will fill the first set of line segments to half-way through the boundary using an even-odd fill, and will thin-stroke the second set of line segments (if present). See the chapter entitled *Draw module* for further details.

Each line segment consists of:

1 bits 0 - 1 = segment type:
0 - terminator (see below)
1 - move to x,y
2 - line to x,y
3 - curve to x1,y1,x2,y2,x3,y3
bits 2 - 4 = x-scaffold link
bits 5 - 7 = y-scaffold link
... coordinates (design units) follow

Terminator:

bit 2 set stroke paths follow (same format, but paths are not closed)
bit 3 set composite character inclusions follow:

Composite character inclusions:

1 character code of character to include (0 => finished)
2/3 x,y offset of this inclusion (design units)

The coordinates are either 8- or 12-bit sign-extended, depending on bit 0 of the character flags (see above), including the composite character inclusions.

The scaffold links associated with each line segment relate to the last point specified in the definition of that move/line/curve, and the control points of a bezier curve have the same links as their nearest endpoint.

1-bpp uncompactd
format

Note that if a character includes another, the appropriate bit in the parent character's chunk dependency flags must be set. This byte tells the Font Manager which extra chunk(s) must be loaded in order to rasterise the parent character's chunk.

1 bit per pixel, bit set => paint in foreground colour, in rows from bottom-left to top-right, not aligned until word-aligned at the end of the character.

The whole character is initially treated as a stream of bits, as for the uncompactd form. The bit stream is then scanned row by row, with consecutive duplicate rows being replaced by a 'repeat count', and alternate runs of black and white pixels are noted. The repeat counts and run counts are then themselves encoded in a set of 4-bit entries.

Bit 2 of the character flags determine whether the initial pixel is black or white (black = foreground), and bits 4-7 are the value of 'f' (see below). The character is then represented as a series of packed numbers, which represent the length of the next run of pixels. These runs can span more than one row, and after each run the pixel colour is changed over. Special values are used to denote row repeats.

<packed number> ==
0 followed by n-1 zeroes, followed by n+1 nibbles
= resulting number + (13-f)*16 + f+1 - 16
i = 1-f i
i = f+1-13 (i-f-1)*16 + next nibble + f + 1
14 followed by n=<packed number> = repeat count of n
15 repeat count of 1 (ie. 1 extra copy of this row)

The optimal value of f lies between 1 and 12, and must be computed individually for each character, by scanning the data and calculating the length of the output for each possible value. The value yielding the shortest result is then used, unless that is larger than the bitmap itself, in which case the bitmap is used.

Repeat counts operate on the current row, as understood by the unpacking algorithm, ie. at the end of the row the repeat count is used to duplicate the row as many times as necessary. This effectively means that the repeat count applies to the row containing the first pixel of the next run to start up.

Note that rows consisting of entirely white or entirely black pixels cannot always be represented by using repeat counts, since the run may span more than one row, so a long run count is used instead.

Music files

Size	Description
8	"Maestro" followed by Return (&0D)
1	2 (type 2 music file)

This is followed zero or more of the following blocks in any order. It is terminated by the end of the file. Note that types 7 to 9 are not implemented in Maestro, but are described for any extensions or other music programs that may be written.

Music data

Size	Description
1	1 indicates Music data follows
4	n = number of 'Gates'.
4*8	length of queue of notes and rests in each channel $L1...L8$ (in bytes), where the data for each note or rest occupies 2 bytes. There are 8 channels, each of which uses a word for the length.
??	<i>For $G=1$ to number of Gates (input above)</i> data for each gate (size depends on type of data) <i>for $C = 1$ to 8 (each channel)</i> <i>for $Q = 1$ to length of note queue in channel C (as above)</i>
$\Sigma L1...L8$	data for note or rest in channel C at point Q in queue

See the full description of gates and the note or rest structures at the end of this type list.

Stave data

Size	Description
1	2 indicates Stave data follows
1	(0 - 3) number of music staves
1	(0 - 1) number of percussion staves

Instrument data

Instrument names are not recorded; only channel numbers.

Size	Description
1	3 indicates Instrument data follows

This is followed by 8 blocks of 2 bytes each:

1	channel number. Always consecutive 1 - 8
1	voice number; 0 = no voice attached

Volume data	Size	Description
	1	4 indicates Volume data follows
	1*8	Volume on each channel = 0 - 7 = ppp - fff. One byte for each channel.
Stereo position data	Size	Description
	1	5 indicates Stereo data follows
	1*8	Stereo position of channel n = 0 - 7 = Full Left...Full Right. One byte for each channel.
Tempo data	Size	Description
	1	6 indicates Tempo data follows
	1	0 - 14, which corresponds to one of: 40, 50, 60, 65, 70, 80, 90, 100, 115, 130, 145, 160, 175, 190, 210 beats per minute
To convert to values to program into SWI Sound_Tempo, use the formula: Sound_Tempo value = Beats per minute * 128 * 4096 / 6000		
Title string	Size	Description
	1	7 indicates title string follows
	n	Null terminated string of n characters total length
Instrument names	Size	Description
	1	8 indicates Instrument names follow
	Σn1...n8	8 null terminated strings for each voice number used in ascending order in command 3 above.
MIDI channels	Size	Description
	1	9 indicates MIDI channel numbers follow
	1*8	MIDI channel number from 1 - 16 on this staff for each channel. 0 indicates not transmitted over MIDI.

Gates

A Gate is a point in the music where something is interpreted. eg. a note, time-signature, key-signature, bar-line or clef can each occupy a gate. The gate data is one byte for a note or rest; 2 bytes for an attribute (time-signature, key-signature, bar-line or clef).

Note or rest

Bit(s)	Description
0 - 7	Gate mask, bit n set to gate 1 note or rest from queue n.

Attribute

Byte	Description
0	0
1	Could be any of the following forms:

Time-signature

Bit(s)	Description
0	1
1 - 4	Number of beats per bar - 1. In the range 0 - 15
5 - 7	Beat type (0 = breve, to 7 = hemidemisemiquaver)

Key-signature

Bit(s)	Description
0 - 1	01 binary
2	0=#, 1=b
3 - 5	0 to 7

Clef

Bit(s)	Description
0 - 2	001 binary
3 - 4	0 = treble, 1 = alto, 2 = tenor, 3 = bass
5 - 6	Stave-1. In the range 0 - 3

Slur

Bit(s)	Description
0 - 3	0001 binary
4	1=on, 0=off
5	Unused
6 - 7	Stave-1. In the range 0 - 3

Octave shift

Bit(s)	Description
0 - 4	00001 binary
5	0=up, 1=down
6 - 7	Stave-1. In the range 0 - 3

Bar

Bit(s)	Description
0 - 5	000001 binary

Reserved for future expansion

Bit(s)	Description
0 - 6	0000001 binary

Notes and rests

Notes and rests are each stored in a 2 byte block that has some common elements.

Notes

Bit(s)	Description
0	Stem orientation. 0 for up and 1 for down
1	Set to 1 to join beams (barbs) to next note
2	Set to 1 to tie with next note
3 - 7	Stave line position 1 - 31 (16 is the centre line)
8 - 10	Accidental
	0 = natural
	1 = sharp
	2 = flat
	3 = double-sharp
	4 = double-flat
	5 = natural sharp
	6 = natural flat
	7 = unused
11 - 12	Number of dots, from 0 - 3
13 - 15	Type. Breve=0 to Hemisemidemi-quaver=7

Rests

Bits	Description
0 - 10	Unused. Set to 0
11 - 12	Number of dots, from 0 - 3
13 - 15	Type. Breve=0 to Hemisemidemi-quaver=7

If a rest coincides with a note, its position is determined by the following note on the same channel.

Table A - VDU codes

List of VDU codes

A list of the VDU codes is given in the table below. Some VDU codes require extra bytes to be sent as parameters; for example, VDU 22 (select screen mode) needs one extra byte to specify the mode. The number of extra bytes needed is also given in the table:

VDU code	Ctrl plus	Extra bytes	Meaning
0	Z or @	0	Do nothing
1	A	1	Send next character to printer only
2	B	0	Enable printer
3	C	0	Disable printer
4	D	0	Write text at text cursor
5	E	0	Write text at graphics cursor
6	F	0	Enable VDU driver
7	G	0	Generate bell sound
8	H	0	Move cursor back one character
9	I	0	Move cursor on one space
10	J	0	Move cursor down one line
11	K	0	Move cursor up one line
12	L	0	Clear text window
13	M	0	Move cursor to start of current line
14	N	0	Turn on page mode
15	O	0	Turn off page mode
16	P	0	Clear graphics window
17	Q	1	Define text colour
18	R	2	Define graphics colour
19	S	5	Define logical colour
20	T	0	Restore default logical colours
21	U	0	Disable VDU drivers
22	V	1	Select screen mode
23	W	9	Multi-purpose command

VDU code	Ctrl plus	Extra bytes	Meaning
24	X	8	Define graphics window
25	Y	5	PLOT
26	Z	0	Restore default windows
27	[0	Do nothing
28	\	4	Define text window
29]	4	Define graphics origin
30	6 or ^	0	Home text cursor
31	- or _	2	Move text cursor

Table B - Modes

List of modes	Mode	Text col x row	Graphics resolution horiz x vert	Colours	Memory used	Monitor types
	0	80 x 32	640 x 256	2	20K	
	1	40 x 32	320 x 256	4	20K	
	2	20 x 32	160 x 256	16	40K	
	3	80 x 25	Text only	4	40K	
	4	40 x 32	320 x 256	2	20K	
	5	20 x 32	160 x 256	4	20K	
	6	40 x 25	Text only	2	20K	
	7	40 x 25	Teletext	16	80K	
	8	80 x 32	640 x 256	4	40K	
	9	40 x 32	320 x 256	16	40K	
	10	20 x 32	160 x 256	256	80K	
	11	80 x 25	640 x 256	4	40K	
	12	80 x 32	640 x 256	16	80K	
	13	40 x 32	320 x 256	256	80K	
	14	80 x 25	640 x 256	16	80K	
	15	80 x 32	640 x 256	256	160K	
	16	132 x 32	1056 x 256	16	132K	
	17	132 x 25	1056 x 256	16	132K	
	18	80 x 64	640 x 512	2	40K	Multi
	19	80 x 64	640 x 512	4	80K	Multi
	20	80 x 64	640 x 512	16	160K	Multi
	21	80 x 64	640 x 512	256	320K	Multi
	23	144 x 56	1152 x 896	2	126K	HRM
	24	132 x 32	1056 x 256	256	264K	
	25	80 x 50	640 x 480	2	37.5K	VGA
	26	80 x 50	640 x 480	4	75K	VGA
	27	80 x 50	640 x 480	16	150K	VGA
	28	80 x 50	640 x 480	256	300K	VGA

Notes:

- 1 Where a monitor type is shown, the mode only generates a useable picture if the computer is connected to a monitor of that type. The types are:
 - Multi: multiple scan-rate
 - HRM: high-resolution mono (if supported by your computer)
 - VGA: includes some VGA monitors with suitable sync connections. Modes suitable for this type can also be used with multiple scan-rate monitors.
- If a monitor type is not shown, the mode will work with a standard monitor.
- 2 Mode 22 is not defined.
- 3 In 256 colour modes, there are some restrictions on the control of the colours. Only 64 base colours may be selected; 4 levels of tinting turn the base colours into 256 shades. Also, the selection from the colour palette of 4096 shades is only possible in groups of 16.
- 4 Mode 3 is a 'gap' mode, where the colour of the gaps is not necessarily the same as the text background.

Recommended modes for the desktop

Of the modes available, the following are the most suitable for a given combination of monitor and number of colours required.

<i>Monitor type</i>	<i>Number of colours</i>	<i>Mode</i>
Standard	16	12
	256	15
Multiscan	16	20
	256	21
VGA	16	27
	256	28
132-column display (Standard/Multiscan)	16	16
	256	24

Table C - File types

List of file types

File types are three-digit hexadecimal numbers. They are divided into three ranges:

E00 - FFF	reserved for use by Acorn
800 - DFF	may be allocated to software houses (A00 to AFF are used for Acornsoft files, 800 to 80C for BBC uniform files)
000 - 7FF	free for users

For each type, there may be a default action on loading and running the file. These actions may change, depending on whether the desktop is in use, and which applications have been seen. The system variables `Alias$@LoadType_XXX` and `Alias$@RunType_XXX` give the actions (XXX = file type).

Some types have a textual equivalent set at startup, which may be used in most commands (but not in the above system variables) instead of the hexadecimal code. These are indicated in the table below by a dagger '†'. For example, file type &FFF is set at startup to have the textual equivalent *Text*. Other textual equivalents may be set as an application starts – for example, Acorn Desktop Publisher sets up file type &AF9 to be *DtpDoc*, and file type &AFA to be *DtpStyle*. These textual equivalents are set using the system variables `File$Type_XXX`, where XXX is the file type.

The following types are currently used or reserved by Acorn. Most file types used by other software houses are not shown. This list may be extended from time to time:

	Type	Description	Textual equivalent	
Acorn file types	FFF	Plain ASCII text	Text	†
	FFE	Command (Exec) file	Command	†
	FFD	Data	Data	†
	FFC	Position independent code	Utility	†

	FFB	Tokenised BASIC program	BASIC	†
	FFA	Relocatable module	Module	†
	FF9	Sprite or saved screen	Sprite	†
	FF8	Absolute application loaded at &8000	Absolute	†
	FF7	BBC font file (sequence of VDU ops)	BBC font	†
	FF6	Fancy font (4 bpp bitmap only)	Font	†
	FF5	PostScript	PostScript	†
	FF4	Dot Matrix data file	DM Data	
	FF3	LaserJet data file	LaserJet	
	FF2	Configuration (CMOS RAM)	Config	
	FF1	Raw unprocessed data (eg terminal streams)	RawData	
	FF0	Tagged Image File Format	TIFF	
	FEF	Diary data	Diary	
	FEE	NotePad data	NotePad	
	FED	Palette data	Palette	†
	FEC	Template file	Template	†
	FEB	Obey	Obey	†
	FEA	Desktop	Desktop	
	FE9	ViewWord	ViewWord	
	FE8	ViewPS	ViewPS	
	FE7	ViewSheet	ViewSht	
	FEO	Desktop accessory	Accessory	
	FDD	Master utilities	MasterUtl	
	F0E	BBC Econet utilities	EconetUtl	
	F09	BBC Winchester utilities	WiniUtil	
Database file types	DB4	SuperCalc III file	SuperCalc	
	DB3	DBase III file	DBaseIII	
	DB2	DBase II	DBaseII	
	DB1	DBase index file	DBaseIndex	
	DB0	Lotus 123 file	Lotus123	
BBC ROM file type	BBC	BBC ROM file (ROMFS)	BBC ROM	†
Acornsoft file types	AFF	Draw file	DrawFile	
	AFE	Mouse event record	Mouse	
	AFD	GCAL source file	Gcal	
	AFC	GCODE intermediate file	GcalOut	
	AFA	DTP style file	DtpStyle	

BBC Uniform file types

AF9	DTP documents	DtpDoc
AF8	First Word Plus file	1stWord+
AF7	Help file	HelpInfo
AF6	ASim trace file	SimTrace
AF5	Mail setup	PostData
AF4	Mail 'Filed'	PostFile
AF3	Mail Postbox	PostBox
AF2	Mail InTray	PostTray
AF1	Maestro file	Music
AF0	ArcWriter file	ARCWriter
ADB	Outline font	New Font
80C	Stationery pad	StationaryPad
80B	Videotex file	VideoTex
80A	Database form file	DataBaseForm
809	Database file	DataBase
808	UniForm PostScript file	UniformPostScript
807	Graphs and charts file	GraphsAndCharts
806	Graphics file	Graphics
805	Drawing file	Drawing
804	Picture file	Picture
803	Spreadsheet file	Spreadsheet
802	UniForm Text only file	UniformText
801	Wordprocessor file	Wordprocessor
800	General BBC UniForm file	Uniform

Table D - Character sets

Introduction

A list of the eight alphabet sets available on your Acorn computer are included in this table. Most are based on the International Standards Organisation ISO 8859 document.

The description of the *Country command in the chapter describing the *International module*, explained the relationship between *country*, *alphabet* and *keyboard*. There are some useful keyboard shortcuts which you can use to switch between alphabets while you are working. You can use these wherever you can use the keyboard: for example, in the Command Line, in Edit, or when entering a filename to save a file. The first two keystroke combinations allow you to switch easily between alphabets.

Alt Ctrl F1 Selects the keyboard layout appropriate to the country UK.

Alt Ctrl F2 Selects the keyboard layout appropriate to the country for which the computer is configured (if available).

Alt <ASCII code typed on numeric keypad>
 Enters the character corresponding to the decimal ASCII number typed.

The following sequence also switches the keyboard layout:

- 1 Press and hold Alt and Ctrl together; press F12.
- 2 Release Ctrl.
- 3 Still holding Alt, type on the numeric keypad the international telephone dialling code for the country you want (eg 044 for Germany, 039 for Italy, 033 for France).
- 4 Release Alt.

Latin1 alphabet (ISO 8859/1)

This is the default alphabet used by Acorn computers.

b ₃	b ₂	b ₁	b ₀	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
0	0	0	0			SP	0	à	P		p			NBSP	°	À	Ð	à	ð
0	0	0	1			!	1	A	Q	a	q			ì	±	Á	Ñ	á	ñ
0	0	1	0			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
0	0	1	1			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
0	1	0	0			\$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
0	1	0	1			%	5	E	U	e	u			¥	µ	Å	Ö	å	ö
0	1	1	0			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
0	1	1	1			'	7	G	W	g	w			§	·	Ç	×	ç	÷
1	0	0	0			(8	H	X	h	x			¨	,	È	Ø	è	ø
1	0	0	1)	9	I	Y	i	y			©	¹	É	Ù	é	ù
1	0	1	0			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
1	0	1	1			+	;	K	[k	{			«	»	Ë	Û	ë	û
1	1	0	0			,	<	L	\	l				¬	¼	Ï	Ü	ï	ü
1	1	0	1			-	=	M]	m	}			¸	½	Í	Ý	í	ý
1	1	1	0			.	>	N	^	n	~			®	¾	Î	Ë	î	ë
1	1	1	1			/	?	O	_	o				¯	¿	Ï	Ë	ï	ÿ

Latin3 alphabet (ISO 8859/3)

				b ₀	b ₁	b ₂	b ₃	b ₄	b ₅	b ₆	b ₇	b ₈	b ₉	b ₁₀	b ₁₁	b ₁₂	b ₁₃	b ₁₄	b ₁₅			
				0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1		
				b ₀	0	0	0	0	1	1	1	1	0	0	0	1	0	1	1	1	1	
				b ₁	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1	
				b ₂	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
				00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15			
b ₀	b ₁	b ₂	b ₃	00				SP	0	à	P	`	p			NBSP	°	À	à			
0	0	0	1	01				!	1	À	Q	a	q			Ë	h	Á	Ñ	á	ñ	
0	0	1	0	02				"	2	B	R	b	r			¸	²	Â	Ô	â	ô	
0	0	1	1	03				#	3	C	S	c	s			£	³	Û	Ó	ó		
0	1	0	0	04				\$	4	D	T	d	t			¤	´	Ä	Ö	ä	ö	
0	1	0	1	05				%	5	E	U	e	u			µ	µ	Ç	Ğ	ç	ğ	
0	1	1	0	06				&	6	F	V	f	v			Ê	ñ	Ĉ	Ö	ĉ	ö	
0	1	1	1	07				'	7	G	W	g	w			Š	·	Ç	×	ç	÷	
1	0	0	0	08				(8	H	X	h	x			"	,	È	Ê	è	ê	
1	0	0	1	09)	9	I	Y	i	y			ı	ı	É	Ù	é	ù	
1	0	1	0	10				*	:	J	Z	j	z			Ş	ş	Ê	Ú	ê	ú	
1	0	1	1	11				+	;	K	L	k	l			Ĝ	ğ	Ë	Û	ë	ü	
1	1	0	0	12				/	<	L	\	l				Ĵ	ĵ	İ	Ü	ı	ü	
1	1	0	1	13				-	=	M]	m	}			ŠHY	½	Í	Û	í	ü	
1	1	1	0	14				.	>	N	~	n	~			Û	Û	Î	Š	î	š	
1	1	1	1	15				/	?	O	_	o				Ž	ž	İ	ß	ï	·	

Latin4 alphabet (ISO 8859/4)

b.				0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1					
b.				0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1				
b.				0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1				
b.				0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1			
				00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15					
b.	b.	b.	b.	00	00	00	00	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
						SP	0	à	P	´	p			NBSP	°	À	Ð	ā	đ					
						!	1	À	Q	á	q				À	á	Á	Ñ	á	ñ				
						"	2	B	R	b	r				κ	À	Õ	â	õ					
						#	3	C	S	c	s				Ŕ	ŕ	Ã	Ɔ	ã	ç				
						\$	4	D	T	d	t				¤	´	Ä	Ö	ä	ö				
						%	5	E	U	e	u				İ	ı	Æ	Ö	æ	ö				
						&	6	F	V	f	v				Ł	ł	Æ	Ö	æ	ö				
						'	7	G	W	g	w				Š	š	ı	×	ı	÷				
						(8	H	X	h	x				"	,	Č	Ø	č	ø				
)	9	I	Y	i	y				Š	š	É	Û	é	Û				
						*	:	J	Z	j	z				Ě	ě	Ě	Ú	ě	ú				
						+	;	K	Ł	k	ł				Ğ	ğ	Ë	Ū	ë	ū				
						/	<	L	\	l					Ʀ	ƣ	È	Ū	è	ū				
						-	=	M	J	m	ı				SHY	Đ	Í	Ū	ı	ū				
						.	>	N	ˆ	n	˜				Ž	ž	Î	Ū	ı	ū				
						/	?	0	_	o					ˉ	Ń	İ	ß	ı	˙				

Greek alphabet (ISO 8859/7)

				0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	SP	0	á	Þ	·	þ			NBSP	°	í	Π	ü	π		
0	0	0	1	!	1	Α	Q	a	q			´	±	A	P	α	ρ		
0	0	1	0	"	2	B	R	b	ῥ			,	²	B	⊗	β	ς		
0	0	1	1	#	3	C	S	c	s			£	³	Γ	Σ	γ	σ		
0	1	0	0	\$	4	D	T	d	t			⊗	,	Δ	T	δ	τ		
0	1	0	1	%	5	E	U	e	u			⊗	,	E	Υ	ε	υ		
0	1	1	0	&	6	F	V	f	v				'	A	Z	Φ	ζ	φ	
0	1	1	1	'	7	G	W	g	w			S	·	H	X	η	χ		
1	0	0	0	(8	H	X	h	x			"	'	E	Θ	Ψ	θ	ψ	
1	0	0	1)	9	I	Y	i	y			©	'	H	I	Ω	ι	ω	
1	0	1	0	*	:	J	Z	j	z			⊗	'	I	K	İ	κ	ϊ	
1	0	1	1	+	;	K	[k	ç			<<	>>	Λ	Ï	λ	ü		
1	1	0	0	,	<	L	\	l				¬	'	O	M	α	μ	ο	
1	1	0	1	-	=	M]	m	ý			SHY	½	N	έ	ν	ύ		
1	1	1	0	.	>	N	^	n	˘			⊗	'	Υ	Ξ	η	ξ	ώ	
1	1	1	1	/	?	O	_	o				-	'	Ω	O	ι	ο	⊗	

Bfont characters

This character set is used in the BBC Master microcomputer. It is retained for the sake of compatibility, but should not be used for new applications.

	0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200	210	220	230	240	250
0	Nothing	Down	Default logical colours	Move text cursor to 00	←	→	←	→	←	→	←	→	←	→	←	→	←	→	←	→	←	→	←	→	←	→
1	Next to printer	Up	Disable VDU	Move text cursor	←	→	←	→	←	→	←	→	←	→	←	→	←	→	←	→	←	→	←	→	←	→
2	Start printer	Clear screen	Select mode	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
3	Stop printer	Start of line	Reprogram characters	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
4	Separate cursors	Typed mode	Define graphics area	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
5	Join cursors	Scroll mode	Par	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
6	Enable VDU	Clear graphics	Default text graphics as ASCII	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
7	Keep	Define text colour	Nothing	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
8	Back	Define graphics colour	Define text area	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
9	Forward	Define logical colours	Define graphics origin	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█

Teletext characters

Teletext alphanumeric

	0	10	20	30	40	50	60	70	80	90	100	110	120
0	Nothing	Down	Nothing	Move cursor to 00	<	2	<	F	F	Z	d	n	x
1	Next to printer	Up	Disable VDU	Move cursor	>	3	=	G	Q	+	e	e	U
2	Start printer	Clear screen	Select mode	■	*	4	>	H	R	l	f	B	Z
3	Stop printer	Start of line	Reprogram characters	!	+	5	?	I	S	+	d	e	U
4	Nothing	Paged mode	Nothing	"	,	6	G	J	T	↑	H	R	
5	Nothing	Scroll mode	Nothing	#	-	7	A	K	U	-	i	s	U
6	Enable VDU	Nothing	Nothing	\$.	8	E	L	V	E	j	t	÷
7	Beep	Nothing	Nothing	%	/	9	C	M	W	a	k	U	Back space and delete
8	Back	Nothing	Nothing	&	0	:	C	N	X	B	l	v	Nothing
9	Forward	Nothing	Nothing	'	1	;	E	O	Y	C	m	w	Alpha red

	130	140	150	160	170	180	190	200	210	220	230	240	250
0	Alpha green	Normal * height	Graphic cyan	█	#	4	>	H	R	W	f	P	Z
1	Alpha yellow	Double height	Graphic white	!	+	5	?	I	S	3	a	o	Q
2	Alpha blue	Nothing	Conceal display	"	,	E	G	J	T	†	r	r	W
3	Alpha magenta	Nothing	Contiguous graphics *	#	-	7	A	K	U	#	i	S	2
4	Alpha cyan	Nothing	Separated graphics	\$.	8	B	L	V	-	j	t	:
5	Alpha * white	Graphic red	Nothing	z	/	9	C	M	W	a	k	U	┌
6	Flash	Graphic green	Black * background	&	0	:	O	N	X	B	L	V	
7	Steady *	Graphic yellow	New background	0	1	;	E	O	Y	E	m	W	
8	Nothing	Graphic blue	Hold graphics	(2	<	F	P	Z	d	n	x	
9	Nothing	Graphic magenta	Release * graphics)	3	=	G	Q	†	e	o	U	

* every line starts with these options

Teletext graphics

	0	10	20	30	40	50	60	70	80	90	100	110	120
0	Nothing	Down	Nothing	Move cursor to 00									
1	Next to printer	Up	Disable VDU	Move cursor -									
2	Start printer	Clear screen	Select mode										
3	Stop printer	Start of line	Reprogram characters										
4	Nothing	Paged mode	Nothing										
5	Nothing	Scroll mode	Nothing										
6	Enable VDU	Nothing	Nothing										
7	Beep	Nothing	Nothing										Back space and delete
8	Back	Nothing	Nothing										Nothing
9	Forward	Nothing	Nothing										Alpha red

	130	140	150	160	170	180	190	200	210	220	230	240	250
0	Alpha green	Normal * height	Graphic cyan										
1	Alpha yellow	Double height	Graphic white										
2	Alpha blue	Nothing	Conceal display										
3	Alpha magenta	Nothing	Contiguous graphics *										
4	Alpha cyan	Nothing	Separated graphics										
5	Alpha * white	Graphic red	Nothing										
6	Flash	Graphic green	Black * background										
7	Steady *	Graphic yellow	New background										
8	Nothing	Graphic blue	Hold graphics										
9	Nothing	Graphic magenta	Release * graphics										

* every line starts with these options

