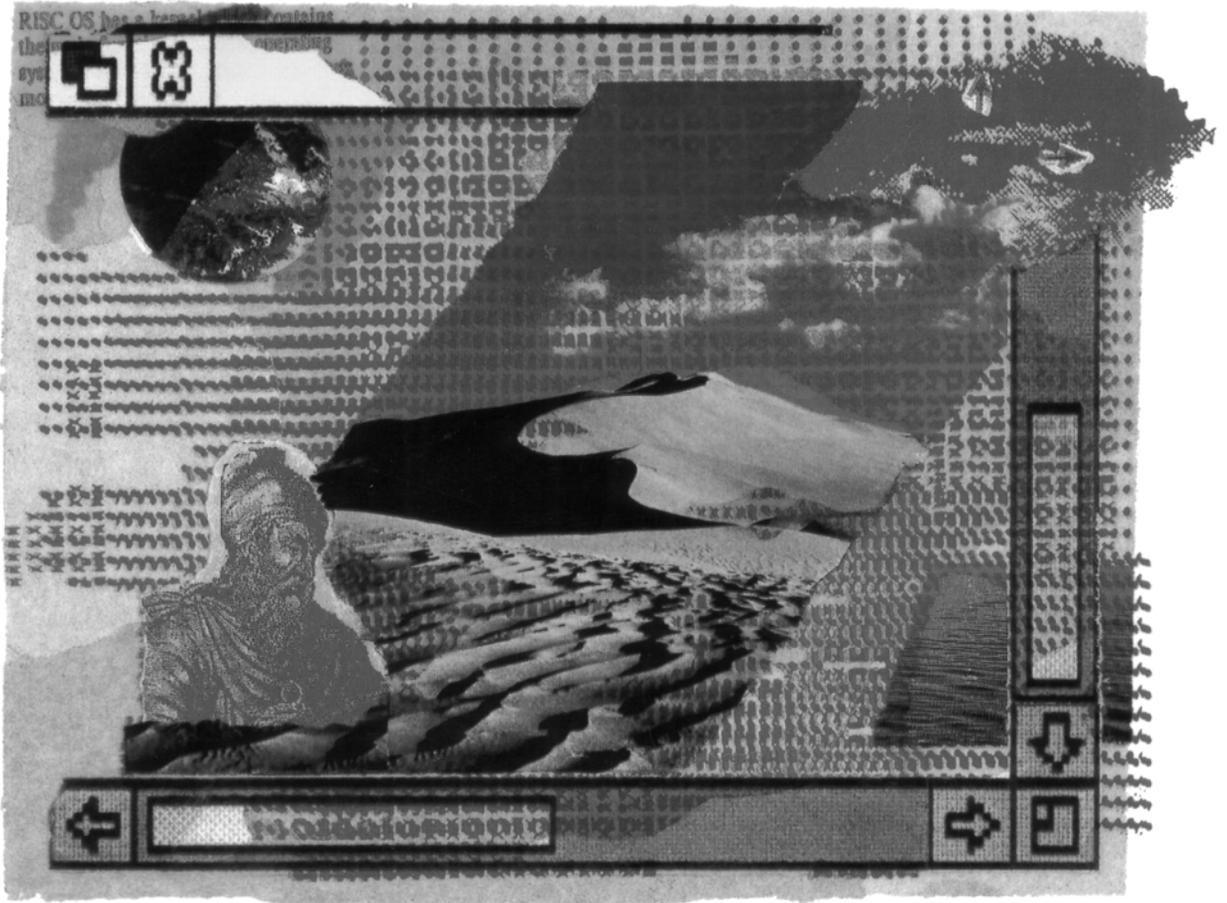


RISC OS 3 Programmer's Reference Manual

Volume 4



Copyright © 1992 Acorn Computers Limited. All rights reserved.

Published by Acorn Computers Technical Publications Department.

No part of this publication may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or stored in any retrieval system of any nature, without the written permission of the copyright holder and the publisher, application for which shall be made to the publisher.

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

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

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

Acorn supplies its products through an international distribution network. Your supplier is available to help resolve any queries you might have.

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

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

Adobe and PostScript are trademarks of Adobe Systems Inc

ARM is a trademark of Advanced RISC Machines Ltd

T_EX is a trademark of the American Mathematical Society

ImageWriter, LaserWriter and Macintosh are trademarks of Apple Computer Inc

DBase is a trademark of Ashton Tate Ltd

UNIX is a trademark of AT&T

Atari is a trademark of Atari Corporation

AutoCAD is a trademark of AutoDesk Inc

Amiga is a trademark of Commodore-Amiga Inc

Commodore is a trademark of Commodore Electronics Limited

SuperCalc is a trademark of Computer Associates

CorelDraw is a trademark of Corel Corporation

VT is a trademark of Digital Equipment Corporation

1st Word Plus is a trademark of GST Holdings Ltd

Deskjet, HP, HPGL, LaserJet and PaintJet are trademarks of Hewlett-Packard

Corporation

Colourjet is a trademark of Integrex Ltd

IBM is a trademark of International Business Machines Corporation

ITC Zapf Dingbats is a trademark of International Typeface Corporation

Helvetica and Times are trademarks of Linotype Corporation

Lotus 123 is a trademark of The Lotus Corporation

MS-DOS is a trademark of Microsoft Corporation

MultiSync and NEC are trademarks of NEC Limited

Epson, EX and FX are trademarks of Seiko Epson Corporation

Sun is a trademark of Sun Microsystems Inc

Ethernet is a trademark of Xerox Corporation

All other trademarks are acknowledged.

Published by Acorn Computers Limited

ISBN for complete set of five volumes: 1 85250 110 3

ISBN for this volume: 1 85250 114 6

Edition 1

Part number 0470,294

Issue 1, 1992

Contents

About this manual 1-xi

Part 1 – Introduction 1-1

An introduction to RISC OS 1-3

ARM Hardware 1-9

An introduction to SWIs 1-23

* Commands and the CLI 1-33

Generating and handling errors 1-41

OS_Byte 1-49

OS_Word 1-59

Software vectors 1-63

Hardware vectors 1-113

Interrupts and handling them 1-119

Events 1-147

Buffers 1-163

Communications within RISC OS 1-181

Part 2 – The kernel 1-201

Modules 1-203

Program Environment 1-289

Memory Management 1-345

Time and Date 1-411

Conversions 1-455

Extension ROMs 1-501

Part 3 – Kernel input/output 1-503

Character Output 1-505

VDU Drivers 1-549

Sprites 1-775

Character Input 1-865

The CLI 1-957

The rest of the kernel 1-969

Part 4 – Using filing systems 2-1

- Introduction to filing systems 2-3
- FileSwitch 2-11
- FileCore 2-197
- ADFS 2-265
- RamFS 2-315
- DOSFS 2-323
- NetFS 2-343
- NetPrint 2-393
- PipeFS 2-413
- ResourceFS 2-415
- DeskFS 2-427
- DeviceFS 2-429
- Serial device 2-445
- Parallel device 2-487
- System devices 2-495
- The Filer 2-499
- Filer_Action and FilerSWIs 2-513
- Free 2-521

Part 5 – Writing filing systems 2-529

- Writing a filing system 2-531
- Writing a FileCore module 2-597
- Writing a device driver 2-607

Part 6 – Networking 2-617

- Econet 2-619
- File server protocol interface 2-705
- The Broadcast Loader 2-739
- BBC Econet 2-741
- Hourglass 2-745
- NetStatus 2-759

Part 7 – The desktop 3-1

- The Window Manager 3-3
- Pinboard 3-291
- Drag A Sprite 3-297
- The Filter Manager 3-301
- The TaskManager module 3-311
- TaskWindow 3-319
- ShellCLI 3-327

Part 8 – Non-kernel input/output 3-331

- ColourTrans 3-333
- The Font Manager 3-411
- SuperSample module 3-529
- Draw module 3-533

Part 9 – Printing 3-563

- Printer Drivers 3-565
- Printer Dumpers 3-673
- PDumperSupport 3-689
- Printer definition files 3-709
- MakePSFont 3-741

Part 10 – Internationalisation 3-743

- MessageTrans 3-745
- International module 3-767
- The Territory Manager 3-793

Part 11 – Sound 4-1

- The Sound system 4-3
- WaveSynth 4-79

Part 12 – Utilities 4-83

- The Buffer Manager 4-85
- Squash 4-103
- ScreenBlank 4-109

Part 13 – Hardware support 4-115

- Expansion Cards and Extension ROMs 4-117
- Floating point emulator 4-169
- ARM3 Support 4-191
- The Portable module 4-205
- Joystick module 4-217

Part 14 – Programmer's support 4-221

- Debugger 4-223
- BASIC and BASICTrans 4-241
- The shared C library 4-249
- Command scripts 4-351

Appendixes and tables 4-359

- Appendix A: ARM assembler 4-361
- Appendix B: Warnings on the use of ARM assembler 4-383
- Appendix C: ARM procedure call standard 4-397
- Appendix D: Code file formats 4-417
- Appendix E: File formats 4-457
- Appendix F: System variables 4-499
- Appendix G: The Acorn Terminal Interface Protocol 4-507
- Appendix H: Registering names 4-549
- Table A: VDU codes 4-555
- Table B: Modes 4-559
- Table C: File types 4-563
- Table D: Character sets 4-567

Part 15 – The kernel 5a-1

- Introduction to RISC OS 3.5 and RISC OS 3.6 5a-3
- ARM hardware 5a-13
- Hardware vectors 5a-21
- Interrupts 5a-33
- Modules 5a-35
- Memory management 5a-37
- CMOS RAM allocation 5a-73
- DMA 5a-81
- Video 5a-101
- JPEG images 5a-145
- Miscellaneous kernel items 5a-163

Part 16 – Filing and networking 5a-165

- FileSwitch 5a-167
- FileCore 5a-171
- ADFS 5a-185
- DOSFS 5a-191
- CDs and CD-ROMs 5a-193
- NetPrint 5a-213
- Parallel and serial device drivers 5a-215
- Keyboard and mouse 5a-231
- Filing system locking and resets 5a-247
- Free 5a-259
- Writing a filing system 5a-261
- Writing a FileCore module 5a-265
- Econet 5a-269
- AUN 5a-279
- The Internet module 5a-305
- Acorn Access 5a-473

Part 17 – The desktop 5a-485

- The desktop 5a-487
- Drag An Object 5a-515
- Draw file renderer 5a-521
- RISC OS boot applications 5a-533
- The colour picker 5a-555
- Printing 5a-579
- Internationalisation 5a-589

Part 18 – Miscellaneous 5a-593

- Sound 5a-595
- CompressJPEG 5a-617
- Expansion card support 5a-625
- Joystick module 5a-637
- Monitor power saving 5a-653
- The Toolbox modules 5a-657

Appendixes 5a-661

- Appendix A: Warnings on the use of ARM assembler 5a-663
- Appendix B: File formats 5a-665
- Appendix C: Errata and omissions for RISC OS 3 PRM 5a-667

Indexes Index-1

Index of * Commands Index-3

Index of OS_Bytes Index-13

Index of OS_Words Index-17

Numeric index of Service Calls Index-19

Alphabetic index of Service Calls Index-25

Numeric index of SWIs Index-31

Alphabetic index of SWIs Index-57

Part 11 – Sound



72 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 physical buffers in main memory are used. These are accessed using four registers in the sound DMA Address Generator (DAG) within the MEMC (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 IOC (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 page 4-10 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 VIDC (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, 25.175MHz or 36MHz, depending on the screen mode and monitor type selected. The default output period is $6\mu\text{s}$, which is compatible with VIDC system clocks running at multiples of 4MHz from 12MHz upwards (ie 12MHz, 16MHz, 20MHz...). This $6\mu\text{s}$ output period is obtained as follows from the 24MHz and 36MHz VIDC system clocks:

- 24MHz clock divided by 144 (6×24)
- 36MHz clock divided by 216 (9×24)

Unfortunately with a VIDC system clock of 25.175MHz (used for VGA screen modes) the same output period cannot be produced. The divider used is the same as for a 24MHz VIDC system clock (ie 144, or 6×24), which results in a slightly shorter output period, and so sounds are approximately a semitone higher.

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 centiseconds, 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.

DMA Buffer format

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.

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 fixup code is never called. (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 VIDC chip.

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.

Sound Channel Control Block (SCCB)

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 loaded from 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 number for attached voice
3	control/status bit flags
4	phase accumulator pitch oscillator
8	phase accumulator timbre oscillator
12	number 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 module), 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_InstallVoice` 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 GateOnCode
12	B GateOffCode
16	B Instantiate
20	B Free
24	LDMFD R13!,{pc}
28	<i>Offset from start of header to voice name</i>

The Fill, Update, GateOn and GateOff entries provide services to fill the DMA buffer at different stages of a note, as detailed in the section entitled *Entry points for buffer filling* on page 4-15.

The Instantiate and Free entries provide facilities to attach or detach the Voice Generator to or from a channel, as detailed in the section entitled *Voice instantiation* on page 4-16.

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 (LDMFD R13!,{pc} above).

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 section entitled *Channel Handler* on page 4-10 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).

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.

Free entry

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.

Service Calls

Service_Sound (Service Call &54)

Parts of the Sound system are starting or dying

On entry

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

R1 = &54 (reason code)

On exit

R0, R1 preserved

Use

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

SWI calls

Sound_Configure (SWI &40140)

Configures the Sound system

On entry

R0 = number of channels, rounded up to 1,2,4 or 8

R1 = sample length (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 length. 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
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 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 (page 4-24), Sound_Volume (page 4-26)

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 maximum right
 -127 for maximum 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 (page 4-20), Sound_Volume (page 4-26)

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

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.

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.

Related SWIs

Sound_Enable (page 4-20), Sound_Speaker (page 4-24)

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 (page 4-28)

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 (page 4-27)

Related vectors

None

Sound_InstallVoice (SWI &40183)

Adds a voice to the Sound system

On entry

R0 = pointer to Voice Generator
R1 = voice slot (0 to install in next free slot, else 1 - 32)

On exit

R0 = pointer to name of previous voice, or null terminated error string if R1 = 0
R1 = voice number allocated, or 0 if unable to install

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 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.

If R0 is in the range 0 - 3, this call takes other action as follows:

R0	Action	Page
0	Reads the name of the voice installed in the specified slot	4-29
1	Adds a voice to the Sound system, specifying its name in the local language	4-29
2	Reads the name of the voice installed in the specified slot, and its local name	4-29
3	Changes the local name of the voice installed in the specified slot	4-29

Related SWIs

Sound_RemoveVoice (page 4-35)

Related vectors

None

Sound_InstallVoice 0 (swi &40183)

Reads the name of the voice installed in the specified slot

On entry

R0 = 0
R1 = voice slot

On exit

R0 = pointer to name of installed voice
R1 preserved

Use

This call reads the name of the voice installed in the specified slot. If the slot is unused RISC OS gives a null pointer. (The Arthur OS gave a pointer to the string ‘*** No Voice’.)

Sound_InstallVoice 1 (SWI &40183)

Adds a voice to the Sound system, specifying its name in the local language

On entry

R0 = 1
R1 = voice slot (0 to install in next free slot, else 1 - 32)
R2 = pointer to Voice Generator
R3 = pointer to voice name in local language, or 0 if no local name

On exit

R0 preserved
R1 = voice number allocated, or 0 if unable to install
R2 = pointer to name of previous voice, or null terminated error string if R1 = 0
R3 preserved

Use

This software interrupt is used by Voice Modules or Libraries to add a Voice Generator to the table of available voices, specifying its name in the local language. 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 reason code is not available in RISC OS 2.

Sound_InstallVoice 2 (swi &40183)

Reads the name of the voice installed in the specified slot, and its local name

On entry

R0 = 2
R1 = voice slot

On exit

R0, R1 preserved
R2 = pointer to name of installed voice
R3 = pointer to name of installed voice in local language

Use

This call reads the name of the voice installed in the specified slot, and its local name. If the slot is unused RISC OS gives a null pointer. (The Arthur OS gave a pointer to the string ‘*** No Voice’.) The local name is otherwise guaranteed to be non-null and valid.

This reason code is not available in RISC OS 2.

Sound_InstallVoice 3 (SWI &40183)

Changes the local name of the voice installed in the specified slot

On entry

R0 = 3
R1 = voice slot
R2 = 0
R3 = pointer to new voice name in local language

On exit

R0 - R3 preserved

Use

This call changes the local name of the voice installed in the specified slot. The local name is set to the new name given, even if it had no local name before this call was made.

This reason code is not available in RISC OS 2.

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 (page 4-29)

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 (page 4-43)

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 (page 4-41), 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 (page 4-41)

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 $R0 \geq \&8000$ 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 (page 4-38)

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 (page 4-37)

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 (page 4-45)

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 (page 4-44)

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 number)

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 (page 4-51)

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 = number 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 Sound_QSchedule must be carefully monitored to observe when overflow occurs.

Related SWIs

Sound_QSchedule (page 4-48)

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

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).

The parameter can be thought of as a hexadecimal fractional number, where the three least significant digits are the fractional part.

Related SWIs

Sound_QBeat (page 4-54)

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 Sound_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 (page 4-53)

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

On or Off

Use

*Audio turns the Sound system on or off. Turning the Sound system off silences it completely, stopping all Sound interrupts and DMA activity. Turning the Sound system back on restores the Sound DMA and interrupt system to the state it was in immediately prior to being turned 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 (page 4-20)

Related vectors

None

*ChannelVoice

Assigns a voice to a channel

Syntax

*ChannelVoice *channel voice_number|voice_name*

Parameters

<i>channel</i>	1 to 8
<i>voice_number</i>	1 to 16, as given by *Voices; or 0 to mute the channel
<i>voice_name</i>	name, as given by *Voices

Use

*ChannelVoice assigns a voice (sound) to one of the eight independent channels used for sound output. It is better to specify the voice by name rather than by number, since the name is independent of the order in which the voices are loaded. Note that the name is case sensitive. Alternatively, you can mute a channel by assigning it a voice slot of 0.

By default, only the first of the eight voices will be available. To make others available, use the SWI Sound_Configure, or enter BASIC and type

>VOICES *n*

where *n* is 2, 4 or 8 (the number of sound channels to enable). Do not, however, confuse the VOICES command in BASIC with *Voices, the command described in this manual.

Example

```
*ChannelVoice 1 StringLib-Pluck
```

Related commands

*Stereo, *Voices

Related SWIs

Sound_Configure (page 4-18), Sound_AttachVoice (page 4-37),
Sound_AttachNamedVoice (page 4-43)

Related vectors

None

*Configure SoundDefault

Sets the configured speaker setting, volume and voice

Syntax

*Configure SoundDefault *speaker volume voice_number*

Parameters

<i>speaker</i>	0 to disable the internal loudspeaker(s) – although the headphones remain enabled 1 to enable the internal loudspeaker(s)
<i>volume</i>	0 (quietest) to 7 (loudest)
<i>voice_number</i>	1 to 16, as given by *Voices

Use

*Configure SoundDefault sets the configured speaker setting, volume and voice. The voice number is assigned to channel 1 only (the default system Bell channel).

Example

```
*Configure SoundDefault 1 7 1
```

Related commands

None

Related SWIs

None

Related vectors

None

Generates a sound after a given delay

Syntax

**QSound channel amplitude pitch duration beats*

Parameters

<i>channel</i>	1 to 8
<i>amplitude</i>	0 (silent) and &FFFF (almost silent) down to &FFF1 (loud) for a linear scale – or &100 (silent) to &17F (loud) for a logarithmic scale, where a change of 16 will halve or double the amplitude
<i>pitch</i>	0 to 255, where each unit represents a quarter of a semitone, with a value of 53 producing middle C – or 256 (&100) to 32767 (&7FFF), where the bottom 12 bits give the fraction of an octave, and the top three bits the octave, with a value of 16384 (&4000) producing middle C
<i>duration</i>	0 to 32767 (&7FFF), giving the duration of the note in twentieths of a second – but a value of 255 (&FF) gives a note of infinite duration (limited by the envelope, if present)
<i>beats</i>	beats delay before the sound is generated, occurring at the rate set by *Tempo

Use

*QSound generates a sound after a given delay. It is identical in effect to issuing a *Sound command after the specified number of beats have occurred. The channel will only sound if at least that number of channels have been selected, and the channel has a voice attached.

Example

```
*QSound 1 &FFF2 &5800 10 50
```

Related commands

*Sound, *Tempo

Related SWIs

Sound_QSchedule (page 4-48)

Related vectors

None

Generates an immediate sound

Syntax

*Sound *channel amplitude pitch duration*

Parameters

<i>channel</i>	1 to 8
<i>amplitude</i>	0 (silent) and &FFFF (almost silent) down to &FFF1 (loud) for a linear scale – or &100 (silent) to &17F (loud) for a logarithmic scale, where a change of 16 will halve or double the amplitude
<i>pitch</i>	0 to 255, where each unit represents a quarter of a semitone, with a value of 53 producing middle C – or 256 (&100) to 32767 (&7FFF), where the bottom 12 bits give the fraction of an octave, and the top three bits the octave, with a value of 16384 (&4000) producing middle C
<i>duration</i>	0 to 32767 (&7FFF), giving the duration of the note in twentieths of a second – but a value of 255 (&FF) gives a note of infinite duration (limited by the envelope, if present)

Use

*Sound generates an immediate sound. The channel will only sound if at least that number of channels have been selected, and the channel has a voice attached.

Example

*Sound 1 &FFF2 &5800 10

Related commands

*QSound

Related SWIs

Sound_ControlPacked (page 4-38), Sound_Control (page 4-41)

Related vectors

None

*Speaker

Turns the internal speaker(s) on or off

Syntax

*Speaker On|Off

Parameters

On or Off

Use

*Speaker turns the internal speaker(s) on or off. It does not affect the 3.5 mm stereo jack socket, which you can still use to play the sound through headphones or an amplifier.

You may still be able to hear a very low level of sound, as this command mutes the speaker(s) rather than totally disabling them.

Example

*Speaker Off

Related commands

*Audio, *Volume

Related SWIs

Sound_Speaker (page 4-24)

Related vectors

None

*Stereo

Sets the position in the stereo image of a sound channel

Syntax

*Stereo *channel position*

Parameters

<i>channel</i>	1 to 8
<i>position</i>	-127(full left) to +127(full right)

Use

*Stereo sets the position in the stereo image of a sound channel.

Example

*Stereo 2 100 *set channel 2 output to come predominantly from the right*

Related commands

*ChannelVoice, *Voices

Related SWIs

Sound_Stereo (page 4-22)

Related vectors

None

*Tuning

Alters the overall tuning of the Sound system

Syntax

*Tuning *relative_change*

Parameters

relative_change -16383 to 16383 (0 resets the default tuning)

Use

*Tuning alters the overall tuning of 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 (page 4-39)

Related vectors

None

*Voices

Displays a list of the installed voices

Syntax

*Voices

Parameters

None

Use

*Voices displays a list of the installed voices by name and number, and shows which voice is assigned to each of the eight channels. A voice can be attached to a channel even if that channel is not currently in use.

Example

```
*Voices
Voice Name
12    1 WaveSynth-Beep
34    2 StringLib-Soft
      3 StringLib-Pluck
      4 StringLib-Steel
      5 StringLib-Hard
56    6 Percussion-Soft
      7 Percussion-Medium
78    8 Percussion-Snare
      9 Percussion-Noise
^^^^^^ Channel Allocation Map
```

Related commands

*ChannelVoice, *Stereo

Related SWIs

Sound_InstallVoice (page 4-29)

Related vectors

None

*Volume

Sets the maximum overall volume of the Sound system

Syntax

*Volume *volume*

Parameters

volume 1 (quietest) to 127 (loudest)

Use

*Volume sets the maximum overall volume of the Sound system. A change of 16 in the volume parameter will halve or double the actual 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. You should therefore use this command sparingly, and only to limit the overall volume of all channels; if a single channel is too loud or soft, you should alter just that channel's volume.

Example

```
*Volume 127
```

Related commands

*Audio, *Configure SoundDefault, *Speaker

Related SWIs

Sound_Volume (page 4-26)

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 *Voice Generators* on page 4-13, 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_RemoveVoice` call, then reinstalled using the `Sound_InstallVoice` 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.

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

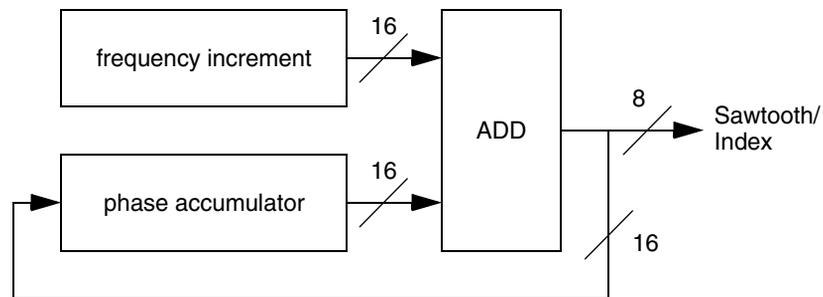
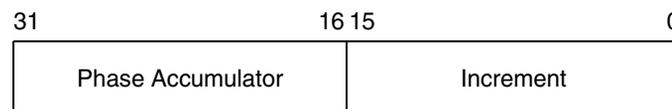


Figure 72.1 Schematic of accumulator/divisor oscillator

Coding

Register field assignment: R_p



ADD $R_p, R_p, R_p, LSL \#16$; 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:

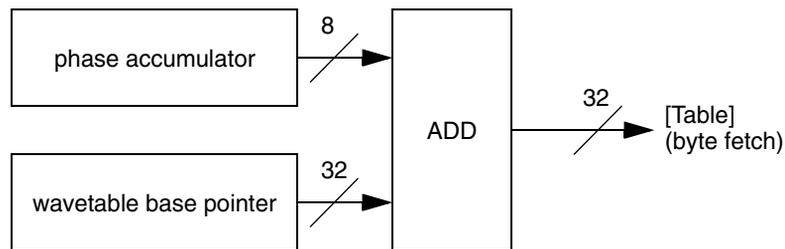


Figure 72.2 Schematic of wavetable access code

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

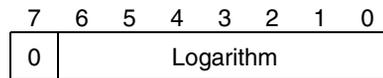
If the overall volume setting changes, then your Update entry point will be called. You can cope with the change in two ways. The first is to re-scale all the values in the wavetable, using the SWI calls `Sound_SoundLog` or `Sound_LogScale`. This has the advantage that buffer filling is faster as the 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.

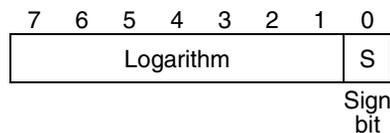
Channel volume

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:

R_a contains amplitude in range 0 to 127

R_s 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
  EQUED VoiceName - VoiceBase
:
.VoiceName EQUUS "WaveVoice"
  EQUB 0
  ALIGN
*****
.LogAmpPtr EQUED 0
.WaveBase EQUED WaveTable%
*****
.Instance ; any instance must use volume scaled log amp table
  STMFD 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-R6} ; 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 centisec 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 1!
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

```

73 WaveSynth

Introduction

WaveSynth is a module that provides a voice generator which is used for the default system bell.

In RISC OS 2 WaveSynth provided a SWI for its own internal use. This has since been removed.

For more information about the use of sound in RISC OS, refer to the chapter entitled *The Sound system* on page 4-3.

Example programs

You can create new wavetables for use with WaveSynth, for example:

```
REM > OrganVoice
OUTFILES$="Organ01"
OUT=OPENOUT OUTFILES$
BPUT#OUT,"!WT:Organ"+STRING$(7,CHR$0);
sizeptr=PTR#OUT
PROCW(0)
FORI%=1TO8:PROCW(8):NEXT
PROCW(13):PROCW(0):PROCW(0)
PROCHDR
size=EXT#OUT
PTR#OUT=sizeptr:PROCW(size)
CLOSE#OUT
REM Pass local name Orgel as parameter on command line
*RMREINIT WAVESYNTH ORGAN01 Orgel
END

DEFPROCW(X%)
LOCALI%
FORI%=1TO4:BPUT#OUT,X%:X%=X%>>8:NEXT
ENDPROC

DEFFNW
RESTORE
DATA 1,1, 0.8,2, 0.6,4, 0.4,8, 0.2,16: REM amplitude,frequency
DATA 0,0
M=0
REPEAT
  READ A$,H$:A=EVALA$
  IF A>0 THEN M+=A
UNTIL A=0
M=&7FFFFFFF/M
RESTORE
B=0
REPEAT
  READ A$,H$:A=EVALA$:H=EVALH$
  IF A>0 THEN B+=FNSIN(A*M,H)
UNTIL A=0
=B

DEFFNSIN(A,F)=A*SIN(F*2*PI*s%/256)
```

```

DEFPROCHDR
MODE0
ORIGIN0,512
MOVE0,0
RESTORE+0
FORI%=1TO14:READJ$:PROCW(EVALJ$):NEXT
PTR#OUT=256
FOR s%=0 TO 255
  B%=FNW
  SYS "Sound_SoundLog",B% TO wave%
  DRAW s%*4,B%>>22
  BPUT#OUT,wave%
NEXT
ENDPROC

REM offset 64 (index 8)
REM descriptor 8 (ATTACK)
DATA &0000007F + (1<<9)
DATA &00090001
REM descriptor 9 (DECAY)
DATA &000000F0 + (31<<9)
DATA &000A0001
REM descriptor 10 (SUS a)
DATA &00000080 + (500<<9)
DATA &000E0001
REM descriptor 11 (SUS b)
DATA &000000DF + (25<<9)
DATA &000A0001
REM descriptor 12 (SUSTAIN)
DATA &00000000 + (&FFFFFF<<9)
DATA &000D0002
REM descriptor 13 (release)
DATA &00000080 + (1<<9)
DATA &000E0001
REM descriptor 14 (Dead)
DATA 0
DATA 0.

```

You can then load the new wavetable into WaveSynth as a module initialisation parameter, eg:

```
REM > Source
obj$="<Obey$Dir>.!RunImage"
DIM MC% 1000,L%-1
FOR I%=8 TO 10 STEP 2
P%=MC%
[OPTI%
.start
MOV R0, #14
ADR R1, instantiation
SWI "XOS_Module"
MOV PC, R14

.instantiation
; Pass local name Orgel as parameter on command line
EQUUS "WaveSynth%Organ <Obey$Dir>.Organ01 Orgel"+CHR$0
]:NEXT
OSCLI "Save "+obj$+" "+STR$~start+" "+STR$~P%
OSCLI "SetType "+obj$+" &FFC"
OSCLI "Stamp "+obj$
```

The facility shown in the above examples for specifying a local name was introduced in RISC OS 3.

Part 12 – Utilities

74 The Buffer Manager

Introduction and Overview

The buffer manager acts as a global buffer managing system, providing a set of calls for setting up a buffer, inserting and removing data from a buffer, and removing a buffer. The buffer manager extends the InsV, RemV and CnpV vector calls to provide access to these buffers and to allow block transfers.

The buffer manager is not available in RISC OS 2.

The buffer manager is used by DeviceFS to provide buffers for the various devices that can be accessed. A device may be linked to a buffer, and may supply routines to be called when data enters the buffer as well as a routine to be called when a buffer is removed (or a new device is attached).

When registering or creating a buffer you can force a specific buffer handle, or request that the buffer manager assign a unique handle. You should note that buffer handles are no longer stored as eight bit quantities.

Block transfers are signalled by setting bit 31 of the buffer handle. Anything you can do on a byte by byte basis you can also do to a block, such as examining the buffer contents.

A number of vectors, events, service calls and UpCalls have been extended or created to enable the buffer manager to function efficiently.

See also the chapter entitled *Buffers* on page 1-163.

Vectors

The SWIs for the buffer manager module allow you to modify the actual buffer itself, but do not supply a way of inserting and removing data from these buffers. Extensions have been made to the following vectors to handle the inserting and removing of data from the buffers, and to allow block inserts. For more details of these vector calls see the chapter entitled *Software vectors* on page 1-63.

- InsV inserts a byte in a buffer
- RemV removes a byte from a buffer
- CnpV counts the number of entries or spaces in a buffer, or purges the contents of a buffer

Events

Because of the above changes to vectors, the following events have been extended so they can indicate that a block transfer occurred. For more details of these events see the chapter entitled *Events* on page 1-147.

- `Event_OutputEmpty` issued when the last character is removed from a buffer
- `Event_InputFull` generated when a character or block is inserted and it failed

Service calls

The service call `Service_BufferStarting` has been added to allow modules which wish to register buffers with the buffer manager to do so. For more details of this service call see page 4-87.

UpCalls

UpCalls are used by the buffer manager to communicate with buffer owners. For more details of these UpCalls see the chapter entitled *Communications within RISC OS* on page 1-181.

- `OS_UpCall 8` issued when data is inserted into the buffer causing the free space to fall below the specified threshold
- `OS_UpCall 9` issued when the free space in the buffer becomes greater than the current threshold.

Service Calls

Service_BufferStarting (Service Call &6F)

Notifies modules that the buffer manager is starting

On entry

R1 = &6F (reason code)

On exit

All registers preserved

Use

This call is passed around modules after the buffer manager has been initialised or reset. Once modules have received this service call they can then register buffers with the buffer manager, and use the Buffer_... SWIs.

SWI calls

Buffer_Create (SWI &42940)

Claims an area of memory from the RMA and registers it as a buffer

On entry

R0 = buffer's flags word:

bit 0: 0 \Rightarrow buffer is dormant, and wake up routine should be called
when data enters it

bit 1: 1 \Rightarrow Event_OutputEmpty should be generated for this buffer

bit 2: 1 \Rightarrow Event_InputFull should be generated for this buffer

bit 3: 1 \Rightarrow UpCalls should be issued when this buffer's free space
threshold is crossed

bits 4 - 31 reserved (should be set to 0 on creation)

R1 = size of buffer to be created

R2 = handle to be assigned to buffer (-1 \Rightarrow get buffer manager to generate handle)

On exit

R0 = buffer handle

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call claims an area of memory from the RMA and registers it as a buffer. If you register a buffer n bytes long, it can hold at most $n - 1$ bytes.

If $R2 = -1$ the buffer manager will attempt to find a unique handle; else the buffer manager will assign the specified handle to the buffer, after checking it is unique.

The buffer's flags word is used to indicate what should happen when data is being inserted and removed from the buffer:

- Bit 0 is set if the buffer is not dormant, and its wake up routine (see the section entitled *The wake up routine* on page 4-97) has been called.
If this bit is clear then the buffer is dormant; when data is then put into the buffer this bit is set and its wake up routine (if any) is called.
- Bit 1 is set if Event_OutputEmpty should be generated for this buffer.
- Bit 2 is set if Event_InputFull should be generated for this buffer.
- Bit 3 is set if UpCalls should be issued when this buffer's free space thresholds are crossed.

Bit 0 should be clear when calling this SWI. Bits 1 - 3 may have any value. The remaining bits are reserved, and should be clear when calling this SWI.

On exit R0 contains the buffer handle being used.

Related SWIs

Buffer_Remove (page 4-90), Buffer_Register (page 4-91), Buffer_LinkDevice (page 4-96)

Related vectors

None

Buffer_Remove (SWI &42941)

Deregisters a buffer and frees its memory

On entry

R0 = handle of buffer to be removed

On exit

R0 preserved

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call attempts to deregister the given buffer. If it succeeds, then any data held by the buffer will be purged, and any future access to the buffer via `InsV`, `RemV` and `CnpV` will be ignored; it will then attempt to free the memory that was claimed for that buffer.

You should only use this call for buffers created and registered using `Buffer_Create`. If you used `Buffer_Register` to register the buffer, you should instead call `Buffer_Deregister` to deregister it.

Related SWIs

`Buffer_Create` (page 4-88), `Buffer_Deregister` (page 4-91), `Buffer_LinkDevice` (page 4-96)

Related vectors

None

Buffer_Register (SWI &42942)

Registers an area of memory as a buffer

On entry

R0 = buffer's flags word:

bit 0: 0 \Rightarrow buffer is dormant, and wake up routine should be called when data enters it

bit 1: 1 \Rightarrow Event_OutputEmpty should be generated for this buffer

bit 2: 1 \Rightarrow Event_InputFull should be generated for this buffer

bit 3: 1 \Rightarrow UpCalls should be issued when this buffer's free space threshold is crossed

bits 4 - 31 reserved (should be set to 0 on registration)

R1 = pointer to start of memory for buffer

R2 = pointer to byte following end of buffer

R3 = handle to be assigned to buffer (-1 \Rightarrow get buffer manager to generate handle)

On exit

R0 = buffer handle

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call registers an area of memory as a buffer. The routine accepts similar parameters to Buffer_Create, but instead of the call claiming the memory for you, you must already have done so yourself, and merely pass the buffer's start and end. If you register a buffer n bytes long, it can hold at most $n - 1$ bytes.

You should not put buffers in the application workspace, as this area of memory might be switched out when someone else tries to access the buffer. However, you can do this if your task will be the only one using the buffer, and it will only be accessed while your task is paged in.

If R3 = -1 the buffer manager will attempt to find a unique handle; else the buffer manager will assign the specified handle to the buffer, after checking it is unique.

The buffer's flags word is used to indicate what should happen when data is being inserted and removed from the buffer:

- Bit 0 is set if the buffer is not dormant, and its wake up routine (see the section entitled *The wake up routine* on page 4-97) has been called.
If this bit is clear then the buffer is dormant; when data is then put into the buffer this bit is set and its wake up routine (if any) is called.
- Bit 1 is set if Event_OutputEmpty should be generated for this buffer.
- Bit 2 is set if Event_InputFull should be generated for this buffer.
- Bit 3 is set if UpCalls should be issued when this buffer's free space thresholds are crossed.

Bit 0 should be clear when calling this SWI. Bits 1 - 3 may have any value. The remaining bits are reserved, and should be clear when calling this SWI.

On exit R0 contains the buffer handle being used.

Related SWIs

Buffer_Create (page 4-88), Buffer_Deregister (page 4-93), Buffer_LinkDevice (page 4-96)

Related vectors

None

Buffer_Deregister (SWI &42943)

Deregisters a buffer

On entry

R0 = handle of buffer to be deregistered

On exit

R0 preserved

Interrupts

Interrupt status is undefined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call attempts to deregister the given buffer. If it succeeds, then any data held by the buffer will be purged, and any future access to the buffer via InsV, RemV and CnpV will be ignored.

You should only use this call for buffers registered using Buffer_Register. If you used Buffer_Create to create and register the buffer, you should instead call Buffer_Remove to deregister it.

Related SWIs

Buffer_Remove (page 4-90), Buffer_Register (page 4-91), Buffer_LinkDevice (page 4-96)

Related vectors

None

Buffer_ModifyFlags (SWI &42944)

Modifies a buffer's flags word

On entry

R0 = handle of buffer to be modified

R1 = EOR mask

R2 = AND mask

On exit

R1 = old value

R2 = new value

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call modifies a buffer's flags word (see page 4-89) by applying an AND mask, followed by an EOR mask. On exit it returns the old and new values of the flags word.

The new value is worked out as follows:

$$new = (old \text{ AND } R2) \text{ EOR } R1$$

You should not modify any reserved bits in the flags word when issuing this call (ie bits 4 - 31 should be set in R2 and clear in R1).

Related SWIs

Buffer_LinkDevice (page 4-96)

Related vectors

None

Buffer_LinkDevice (SWI &42945)

Links a set of routines to the specified buffer

On entry

R0 = buffer handle

R1 = pointer to routine to call when data enters the dormant buffer (0 ⇒ none)

R2 = pointer to routine to call when owner of buffer is to change (0 ⇒ cannot be changed)

R3 = private word to be passed to above routines

R4 = pointer to workspace for above routines

On exit

R0 - R4 preserved

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call links a set of routines to the specified buffer.

The routines are called with the same entry conditions. The processor may be in any mode and interrupt state. The registers are as follows:

On entry

R0 = buffer handle

R8 = private word (as specified in R3)

R12 = pointer to workspace for routine (as specified in R4)

Such routines are typically used to wake up devices attached to a previously dormant buffer so they can start processing data that has appeared, and to shutdown a device when another wishes to access its buffer. In particular, DeviceFS uses this mechanism.

The wake up routine

R1 contains a pointer to a routine to be called when data enters the buffer and it is currently marked dormant. Before calling this ‘wake up’ routine, the buffer manager first sets bit 0 in the buffer’s flags word, marking it as no longer dormant. On exit from the wake up routine you must preserve the entire state of the processor: ie the register contents (including the PSR), the mode, and the state of IRQ and FIQ.

If this pointer (ie R1) is zero, the buffer manager does not attempt to call a wake up routine for the specified buffer.

The owner change routine

R2 contains a pointer to a routine to be called whenever the owner of the buffer is about to change. This occurs:

- when an attempt is made to remove or deregister the buffer by calling `Buffer_Remove` or `Buffer_Deregister` respectively
- when an attempt is made to link to the buffer by another call of this SWI for the same buffer
- when an attempt is made to kill the buffer manager.

On return from this ‘owner change’ routine you can return an error in the usual way (V set, R0 points to an error block) and thus halt the attempt to change the buffer’s owner; you’ll also – coincidentally – halt whatever caused the attempt. For example, this SWI may sometimes fail because the given buffer may already have an owner that is refusing to detach itself. If you don’t return an error you must preserve the entire state of the processor: ie the register contents (including the PSR), the mode, and the state of IRQ and FIQ.

If this pointer (ie R2) is zero, the buffer manager will always return an error if an attempt is made to change the buffer’s owner.

Related SWIs

`Buffer_Remove` (page 4-90), `Buffer_Deregister` (page 4-93), `Buffer_ModifyFlags` (page 4-94), `Buffer_UnlinkDevice` (page 4-98)

Related vectors

None

Buffer_UnlinkDevice (SWI &42946)

Unlinks a set of routines from the specified buffer

On entry

R0 = buffer handle

On exit

R0 preserved

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call unlinks all routines that were previously linked to the specified buffer by calling Buffer_LinkDevice. No warning is given of this (ie the buffer's change owner routine is not called), and any data that is currently stored within the buffer is purged.

You should only make this call if it was you that initially linked the routines; anyone else calling this SWI could confuse the system.

Related SWIs

Buffer_LinkDevice (page 4-96)

Related vectors

None

Buffer_GetInfo (swi &42947)

Returns data about the buffer

On entry

R0 = buffer handle

On exit

R0 = buffer's flags word

R1 = pointer to start of buffer in memory

R2 = pointer to byte following end of buffer

R3 = offset within buffer of insertion point

R4 = offset within buffer of removal point

R5 = remaining free space in buffer

R6 = number of characters in buffer

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call returns data about the buffer: its flags word, position in memory, the offsets within the buffer of its insertion and removal points, the amount of free space, and the number of characters in the buffer.

The insertion and removal points wrap around from the end of the buffer to the start, so you should not assume that the insertion point's offset will be greater than that of the removal point. Furthermore, you should not assume that the sum of R5 and R6 (the free space in the buffer and the number of characters in the buffer) will be the same as the size of the buffer.

Related SWIs

None

Related vectors

None

Buffer_Threshold (SWI &42948)

Sets or reads the warning threshold of the buffer

On entry

R0 = buffer handle
R1 = threshold (0 = none, -1 to read)

On exit

R1 = previous value

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 to set or read the warning threshold of the buffer. UpCalls are issued if bit 3 of the buffer's flags word is set, and the amount of free space in the buffer crosses this threshold value. For details of the UpCalls see the chapter entitled *Communications within RISC OS* on page 1-181.

Related SWIs

Buffer_Create (page 4-88), Buffer_Register (page 4-91)

Related vectors

None



Introduction and Overview

This module provides general compression and decompression facilities of a lossless nature through a SWI interface. The algorithm is 12-bit LZW, however, this may change in future releases.

The interface is designed to be restartable, so that compression or decompression can occur from a variety of locations. Operations involving file I/O can easily be constructed from the operations provided.

This module is not available in RISC OS 2.

The module is used by the Squash application to generate files of type Squash (&FCA). The format of these files is documented in the section entitled *Squash files* on page 4-497.

Errors

The following errors can be returned by the Squash module:

Error number	Error text
&921	Bad address for module Squash
&922	Bad input for module Squash
&923	Bad workspace for module Squash
&924	Bad parameters for module Squash

SWI calls

Squash_Compress (SWI &42700)

Provides general compression of a lossless nature

On entry

R0 = flags:

- bit 0: 0 \Rightarrow start new operation, 1 \Rightarrow continue existing operation (using existing workspace contents)
- bit 1: 0 \Rightarrow end of the input, 1 \Rightarrow more input after this
- bit 2: reserved (must be zero)
- bit 3: 0 \Rightarrow no effect, 1 \Rightarrow return the work space size required and the maximum output size in bytes (all other bits must be 0)
- bits 4 - 31 reserved (must be zero)

R1 = input size (-1 \Rightarrow do not return maximum output size) – if bit 3 of R0 is set; or workspace pointer – if bit 3 of R0 is clear

R2 = input pointer – if bit 3 of R0 is clear

R3 = number of bytes of input available – if bit 3 of R0 is clear

R4 = output pointer – if bit 3 of R0 is clear

R5 = number of bytes of output space available – if bit 3 of R0 is clear

On exit

R0 = required work space size – if bit 3 of R0 set on input; else output status – if bit 3 of R0 clear on input:

- 0 \Rightarrow operation completed
- 1 \Rightarrow operation ran out of input data (R3 = 0)
- 2 \Rightarrow operation ran out of output space (R5 < 12)

R1 = maximum output size (-1 \Rightarrow don't know or wasn't asked) – if bit 3 of R0 set on input; else preserved – if bit 3 of R0 clear on input

R2 updated to show first unused input byte – if bit 3 of R0 clear on input

R3 updated to show number of input bytes not used – if bit 3 of R0 clear on input

R4 updated to show first unused output byte – if bit 3 of R0 clear on input

R5 updated to show number of output bytes not used – if bit 3 of R0 clear on input

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This call provides general compression of a lossless nature. It acts as a filter on a stream of data. The call returns if either the input or the output is exhausted.

It is recommended that you use the following facility to determine the maximum output size rather than attempting to calculate it yourself:

Call the SWI first with bit 3 of R0 set and the input size placed in R1. The maximum output size is then calculated and returned on exit in R1. You can use this value to allocate the required amount of space and call the SWI again setting the registers as appropriate.

If for any reason the SWI cannot calculate the maximum output size it will return -1 in R1.

The workspace size required is returned in R0.

The algorithm used by this module is 12-bit LZW, as used by the UNIX 'compress' command (with -b 12 specified). If future versions of the module use different algorithms, they will still be able to decompress existing compressed data.

If bits 0 and 1 of R0 are clear, and the output is definitely big enough, a fast algorithm will be used.

The performance of compression on an 8Mhz A420 with ARM2 is approximately as follows:

Store to store	Fast case
24 Kbytes per second	68 Kbytes per second

where *Fast case* is store to store, with all input present, and with an output buffer large enough to hold all output.

Related SWIs

Squash_Decompress (page 4-106)

Related vectors

None

Squash-Decompress (SWI &42701)

Provides general decompression of a lossless nature

On entry

R0 = flags:

- bit 0: 0 \Rightarrow start new operation, 1 \Rightarrow continue existing operation (using existing workspace contents)
- bit 1: 0 \Rightarrow end of the input, 1 \Rightarrow more input after this
- bit 2: 0 \Rightarrow normal, 1 \Rightarrow you may assume that the output will all fit in this buffer (allows a faster algorithm to be used, if bits 0 and 1 are both 0)
- bit 3: 0 \Rightarrow no effect, 1 \Rightarrow return the work space size required and the maximum output size in bytes (all other bits must be 0)
- bits 4 - 31 reserved (must be zero)

R1 = input size (-1 \Rightarrow do not return maximum output size) – if bit 3 of R0 is set; or workspace pointer – if bit 3 of R0 is clear

R2 = input pointer – if bit 3 of R0 is clear

R3 = number of bytes of input available – if bit 3 of R0 is clear

R4 = output pointer – if bit 3 of R0 is clear

R5 = number of bytes of output space available – if bit 3 of R0 is clear

On exit

R0 = required work space size – if bit 3 of R0 set on input; else output status – if bit 3 of R0 clear on input:

- 0 \Rightarrow operation completed
- 1 \Rightarrow operation ran out of input data (R3 < 12)
- 2 \Rightarrow operation ran out of output space (R5 = 0)

R1 = maximum output size (-1 \Rightarrow don't know or wasn't asked) – if bit 3 of R0 set on input; else preserved – if bit 3 of R0 clear on input

R2 updated to show first unused input byte – if bit 3 of R0 clear on input

R3 updated to show number of input bytes not used – if bit 3 of R0 clear on input

R4 updated to show first unused output byte – if bit 3 of R0 clear on input

R5 updated to show number of output bytes not used – if bit 3 of R0 clear on input

Interrupts

Interrupt status is undefined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI provides general decompression of a lossless nature.

Note: The current algorithm cannot predict what the size of the decompressed output will be. This means that, currently, -1 is always returned on exit in R1. In future releases this may change; it is therefore recommended that you call the SWI first with bit 3 of R0 set and the input size placed in R1.

If R1 is not equal to -1 then you can use this value to allocate the required amount of space and call the SWI again, setting the registers as appropriate. If R1 is equal to -1 you must attempt to calculate the maximum output size yourself.

The workspace size required is returned in R0.

In the case where $R3 < 12$, the unused input must be resupplied.

The performance of decompression on an 8Mhz A420 with ARM2 is approximately as follows:

Store to store	Fast case
48 Kbytes per second	280 Kbytes per second

where *Fast case* is store to store, with all input present, and with an output buffer large enough to hold all output.

Related SWIs

Squash_Compress (page 4-104)

Related vectors

None



76 ScreenBlank

Introduction and Overview

The ScreenBlank module provides the facilities needed to support screen blanking. There are two service calls so that applications can tell when the screen is blanked and when it is restored.

There is also a * Command with which you can override the default time of inactivity before the screen blanks. The default time itself is set using the Configure application; there is no defined programmers' interface to do so.

The ScreenBlank module also provides a SWI for internal use by the Portable module; you must not use it in your own code.

This module is not available in RISC OS 2.

Service Calls

Service_ScreenBlanked (Service Call &7A)

Screen blanked by screen blanker

On entry

R1 = &7A (reason code)

On exit

All registers must be preserved.

Use

This service call is issued by the screen blanker, after the screen has been blanked This service call should not be claimed.

Service_ScreenRestored (Service Call &7B)

Screen restored by screen blanker

On entry

R0 = 0, or flags passed in R4 to ScreenBlanker_Control 2

R1 = &7B (reason code)

On exit

All registers must be preserved.

Use

This service call is issued by the screen blanker, after the screen has been restored. This service call should not be claimed.

R0 is normally zero. If however the call results from a flash cycle, then it will be set to the value of R4 that was passed to ScreenBlanker_Control 2.

SWI calls

ScreenBlanker_Control (SWI &43100)

This SWI is for internal use by the Portable module. You must not use it in your own code.

* Commands

*BlankTime

Sets the time of inactivity before the screen blanks

Syntax

```
*BlankTime [W|O] [time]
```

Parameters

W	writing to the screen finishes screen blanking
O	writing to the screen does not finish screen blanking
<i>time</i>	time of inactivity before the screen blanks

Use

*BlankTime sets the time in seconds before the screen blanks. If, during this time, there is no activity (ie no keyboard or mouse input is received, and – with the W option – there is no writing to the screen) the screen then blanks. This saves ‘burn in’ on the phosphor of your monitor, which occurs when the monitor consistently displays a particular image, such as the desktop.

Screen blanking finishes as soon as there is activity (see above).

If no option is specified, O is assumed.

The blank time is only retained until the next reset.

Example

```
*BlankTime W 600      blanks the screen if neither input nor output occur for 10
                        minutes
```

Related commands

None

Related SWIs

None

Related vectors

WrchV (claimed by W option)

Part 13 – Hardware support



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).

Extension ROMs are ROMs fitted in addition to the main ROM set, which provide software modules which are automatically loaded by RISC OS on power-on. Note that **RISC OS 2 does not support extension ROMs**. Extension ROMs are provided so that Acorn can add extra modules to RISC OS, or provide replacement modules for those already in RISC OS. **You must not use them.**

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 expansion cards for RISC OS to communicate with them; in short, all you need to know to write their software. For completeness, it gives the same information for extension ROMs; but – of course – this is irrelevant to you, as you shouldn't use extension ROMs.

The two topics are covered together because both use substantially the same layout of code and data, and the same SWIs. For more details on writing modules, see the chapter entitled *Modules* on page 1-203.

One thing this chapter does not tell you is how to design expansion card 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.

Some RISC OS computers can also support extension ROMs. The availability, size and number of extension ROM sockets depends on which type of RISC OS computer you are using. For example, the A5000 has a single socket for an 8 bit wide ROM.

Software

Expansion cards

Expansion cards can have some or all of the following software included:

- an Expansion Card Identity, to give RISC OS information about the card (see page 4-120 and page 4-122)
- Interrupt Status Pointers, to tell RISC OS where to look to find out if the card is generating interrupts (see page 4-127)
- a Chunk Directory, that defines what separate parts of the card's memory space are used for (see page 4-128)
- a Loader, to access paged memory held outside the card's address space (see page 4-130)

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.

Extension ROMs

Extension ROMs must include the following software:

- an Extension ROM Header, to give RISC OS information about the ROM and to differentiate it from an expansion card (see page 4-119)
- an Extended Expansion Card Identity, to give RISC OS information about the ROM (see page 4-122)
- null Interrupt Status Pointers, because a ROM cannot generate interrupts (see page 4-127)
- a Chunk Directory, that defines what each part of the ROM's memory space is used for (see page 4-128).

Technical Details

In general, RISC OS recognises extension ROMs or ROM sets which are 8, 16 or 32 bits wide, provided the ROM adheres to the specification below. 32 bit wide extension ROM sets are directly executable in place, saving on user RAM. 8 or 16 bit wide sets have to be copied into RAM to execute.

An extension ROM set must end on a 64K boundary or at the start of another extension ROM. This is normally not a problem as it is unlikely you would want to use a ROM smaller than a 27128 (16K), and the normal way of addressing this would mean that the ROM would be visible in 1 byte out of each word, ie within a 64K addressable area.

Extension ROM Headers

Extension ROMs must have a 16 byte *Extension ROM Header* at the **end** of the ROM image, which indicates the presence of a valid extension ROM. The ‘header’ is at the end because RISC OS scans the ROM area downwards from the top.

For a ROM image of size n bytes, the format of the header at the end is as follows:

Byte address	Contents																								
$n-16$	1-word field containing n																								
$n-12$	1-word checksum (bottom 32 bits of the sum of all words from addresses 0 to $n-16$ inclusive)																								
$n-8$	2-word id ‘ExtnROM0’ indicating a valid extension ROM, ie: <table> <tbody> <tr> <td>$n-8$</td> <td>&45</td> <td>‘E’</td> </tr> <tr> <td>$n-7$</td> <td>&78</td> <td>‘x’</td> </tr> <tr> <td>$n-6$</td> <td>&74</td> <td>‘t’</td> </tr> <tr> <td>$n-5$</td> <td>&6E</td> <td>‘n’</td> </tr> <tr> <td>$n-4$</td> <td>&52</td> <td>‘R’</td> </tr> <tr> <td>$n-3$</td> <td>&4F</td> <td>‘O’</td> </tr> <tr> <td>$n-2$</td> <td>&4D</td> <td>‘M’</td> </tr> <tr> <td>$n-1$</td> <td>&30</td> <td>‘0’</td> </tr> </tbody> </table>	$n-8$	&45	‘E’	$n-7$	&78	‘x’	$n-6$	&74	‘t’	$n-5$	&6E	‘n’	$n-4$	&52	‘R’	$n-3$	&4F	‘O’	$n-2$	&4D	‘M’	$n-1$	&30	‘0’
$n-8$	&45	‘E’																							
$n-7$	&78	‘x’																							
$n-6$	&74	‘t’																							
$n-5$	&6E	‘n’																							
$n-4$	&52	‘R’																							
$n-3$	&4F	‘O’																							
$n-2$	&4D	‘M’																							
$n-1$	&30	‘0’																							

Extension ROM width

Note that this header will not necessarily appear in the memory map in the last 16 bytes if the ROM set is 8 or 16 bits wide. In the 8-bit case, the header will appear in one of the four byte positions of the last 16 words, and in the 16-bit case, in one of the two half-word positions of the last 8 words. However, RISC OS copes with this, and uses the mapping of the ID field into memory to automatically derive the width of the extension ROM.

Introduction to Expansion Card Identities

Expansion cards

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 (see below)
- an extended ECId of eight bytes, which may be followed by other information (see page 4-122).

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.

Extension ROMs

As well as the Extension ROM header at the end of the ROM image, Extension ROMs must also have a header at the **start** of the ROM image. This header is identical in format to an Extended Expansion Card Identity, and is present for the use of the Expansion Card Manager, which handles much of the extension ROM processing. See page 4-122 onwards, paying particular attention to the section entitled *Mandatory values for extension ROMs*.

Simple Expansion Card Identity

Expansion cards can use a simple ECId, which is one byte long. You should only use one for the very simplest of expansion cards, or temporarily during development.

- Most expansion cards should instead implement the extended ECId, which eliminates the possibility of expansion card IDs clashing.
- Extension ROMs must use an extended ECId, rather than a simple ECId.

Restrictions imposed by a Simple ECId

If you do use a simple ECId, your expansion card **must** be 8 bits wide. The only operations that you may perform on its ROM are Podule_RawRead (see page 4-151) or Podule_RawWrite (see page 4-153).

Format of a simple ECId

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 (0	ID field extended ECId used)
FIQ	0	not requesting FIQ
	1	requesting FIQ
IRQ	0	not requesting IRQ
	1	requesting IRQ

Acorn conformance bit (A)

This bit must be zero for expansion cards that conform to this Acorn specification.

ID field (ID [3:0])

If you are using a simple ECId, the four 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 (IRQ and FIQ)

The interrupt status bits are discussed below in the section entitled *Generating interrupts from expansion cards* on page 4-126.

Expansion card presence (bit 1)

This must be zero, as shown above. For more information, see the section entitled *Expansion card and extension ROM presence* on page 4-125.

Extended Expansion Card Identity

An expansion card's ECID is extended if the ID field of its ECID low byte is zero. 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 the eight bytes defined below.

Expansion card width

If an expansion card has an extended ECID, the first 16 bytes of its address space are always assumed to be byte-wide. These 16 bytes contain the 8 byte extended ECID itself, and a further 8 bytes (typically the Interrupt status pointers – see below). 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).

Current restrictions

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.

Format of an extended ECId

The format of an extended ECId is as follows:

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 pointers
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 (C[7:0])

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 (M[15:0])

Every expansion card should have a code for manufacturer. If you have not already been allocated one, you should consult Acorn.

Product type code (P[15:0])

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 (R)

Reserved fields must be set to zero to cater for future expansion.

Width field (W[1:0])

This field must currently be set to zero (expansion card is 8 bits wide). For more information, see the earlier section entitled *Expansion card width* on page 4-122.

Interrupt Status Pointers presence (IS)

See the sections entitled *Generating interrupts from expansion cards* on page 4-126, and *Interrupt Status Pointers* on page 4-127.

Chunk directory presence (CD)

See the section entitled *Chunk directory structure* on page 4-128.

Acorn conformance bit (A)

This bit must be zero for expansion cards that conform to this Acorn specification.

ID field (bits 6 - 3 of low byte)

If you are using an extended ECId, these bits must be zero, as shown above. A non-zero value shows that you are instead using a simple ECId; for more information see page 4-121.

Interrupt status bits (IRQ and FIQ)

The interrupt status bits are discussed below in the section entitled *Generating interrupts from expansion cards* on page 4-126.

Expansion card presence (bit 1 of low byte)

This must be zero, as shown above. For more information, see the section entitled *Expansion card and extension ROM presence* on page 4-125.

Mandatory values for extension ROMs

An extension ROM must include an extended ECId. This starts at the bottom of the ROM image, and consists of eight bytes as defined above.

For an extension ROM, certain fields within the extended ECId must have particular values:

- The product type code must be &87 (ie the product type is an extension ROM).
- The width field must always be 0 (8 bits wide), irrespective of the ROM's actual width, which RISC OS automatically derives (see the section entitled *Extension ROM width* on page 4-119).

Because the width field does not vary, you do not need to change the image of an extension ROM if you change the width of ROM in which it is placed.

- Both the Interrupt Status Pointer field and the Chunk Directory field must be 1, showing the ECId is followed by Interrupt Status Pointers, then by a Chunk Directory.
- The Acorn conformant field must be 0, to show that the extension ROM is Acorn conformant.
- The interrupt status bits (FIQ and IRQ) must both be clear, to show that the extension ROM is not requesting an interrupt.

Expansion card and extension ROM presence

All expansion cards and extension ROMs **must have bit 1 low** in the low byte of an ECId (whether simple or extended), so that RISC OS can tell if there are any of them 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 an expansion card is present, and the ECId is mapped into memory (which it must be immediately after a reset), the bit will instead be clear.

Generating interrupts from expansion cards

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

If an expansion card has an extended ECId, you must set the IS bit of the ECId and provide *Interrupt Status Pointers* (see below) if either of the following applies:

- you are also using Chunk Directories (see below)
- you want to relocate the interrupt status bits from the low byte of the ECId.

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).

Finding out more

To find out more about generating interrupts from expansion cards under RISC OS, you can:

- see the chapters entitled *ARM Hardware* on page 1-9 and *Interrupts and handling them* on page 1-119.
- consult the *Acorn RISC Machine family Data Manual*. VLSI Technology Inc. (1990) Prentice-Hall, Englewood Cliffs, NJ, USA: ISBN 0-13-781618-9.
- consult the datasheets for any components you use
- contact Customer Support and Services for further hardware-specific details.

Interrupt Status Pointers

Expansion cards

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
IRQ Status Bit position mask	&24
	&20

The 24-bit address field must contain a 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 FIQ or IRQ

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.

Extension ROMs

Extension ROMs must have a Chunk Directory, hence they must also provide Interrupt Status Pointers. However, extension ROMs generate neither FIQ nor IRQ; consequently their Interrupt Status Pointers always consist of eight zero bytes.

Chunk directory structure

If the CD bit of an 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* follow the Interrupt Status Pointers.

The chunks of data and/or code are stored in the expansion card's ROM, or in the extension ROM.

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 address 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 location pointed to contains the ASCII string of the information.)
		7 Ethernet binary ID (length is always 6 bytes)

8	PCB revision (length is always 4 bytes, treated as a word)
9 - 14	reserved
15	empty chunk

Those Chunks with OS[2:0] = 7, are operating system independent and are mostly 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 a 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.

An example program used with an expansion card is shown at the end of this chapter. Its output is a file suitable for programming into a PROM or an EPROM.

Expansion card 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 for an expansion card

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 E59FA0E4 Reset LDR r10,=2_00000011111111111111100000000000
14 E00BA00A AND r10,r11,r10 ; Get hardware address from combined one
18 E28AAA03 ADD r10,r10,#PageReg
1C E3A02000 MOV r2,#0
20 E4CA2000 STRB r2,[r10]
24 E3DEF201 BICS pc,lr,#V_bit
28 E59F40C4 ReadByte LDR r3,=2_0000001111111111111111100000000000
2C E00B4004 AND r3,r11,r3 ; Get hardware address from combined one
30 E284AA03 ADD r10,r3,#PageReg
34 E3510B3E CMP r1,#&F800 ; Last page
38 228F0048 ADRHS r0,ErrorATB
3C 239EF201 ORRHSS pc,lr,#V_bit
40 E2812B02 ADD r2,r1,#1 :SHL: PageSize
44 E1A025C2 MOV r2,r2,ASR #PageSize
48 E4CA2000 STRB r2,[r10]
4C E3C12BFE BIC r2,r1,#&7F :SHL: PageSize
50 E7D40102 LDRB r0,[r3,r2,ASL #2] ; Word addressing
54 E3DEF201 BICS pc,lr,#V_bit
58 E28F0000 WriteByte ADR r0,ErrorNW
5C E39EF201 ORRS pc,lr,#V_bit
60 00000580 ErrorNW DCD ErrorNumber_NotWriteable
64          DCB ErrorString_NotWriteable,0
92 00 00          ALIGN
94 00000584 ErrorATB DCD ErrorNumber_AddressTooBig
98          DCB ErrorString_AddressTooBig,0
AC          END

```

The bit masks are used to separate the fields of a combined hardware address – see the description of `Podule_HardwareAddress` (page 4-155) 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_EnumerateChunks` 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` (page 4-155) or `Podule_HardwareAddresses` (page 4-159).

ROM sections

Most of the SWIs provided by the Expansion Card Manager take a ROM section as a parameter. This identifies the expansion card or extension ROM upon which the command acts. ROM sections used by RISC OS are:

ROM section	Meaning	
-1	System ROM	
0	Expansion card 0	
1	Expansion card 1	
2	Expansion card 2	
3	Expansion card 3	
-2	Extension ROM 1	(not in RISC OS 2)
-3	Extension ROM 2	(not in RISC OS 2)
-4	Extension ROM 3 (etc)	(not in RISC OS 2)

None of the SWIs described in this chapter will act upon the system ROM.

'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.

Service Calls

Service_PreReset (Service Call &45)

Pre-reset

On entry

R1 = &45 (reason code)

On exit

R1 preserved to pass on (do not claim)

Use

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

Service_ADFSPodule (Service Call &10800)

Issued by ADFS to locate an ST506 expansion card

On entry

R1 = &10800 (reason code)
R2 = address of current ST506 hard disc controller
R3 = address of IRQ status register for current hard disc controller
R4 = mask which, when ANDed with IRQ status register, gives non-zero value if
 IRQs are enabled
R5 = address of IRQ mask register for current hard disc controller
R6 = mask which, when ORd with IRQ mask register, enables IRQ

On exit

All registers preserved to pass on, else:

R1 = 0 to claim
R2 = address of new ST506 hard disc controller
R3 = address of IRQ status register for new hard disc controller
R4 = mask which, when ANDed with IRQ status register, gives non-zero value if
 IRQs are enabled
R5 = address of IRQ mask register for new hard disc controller
R6 = mask which, when ORd with IRQ mask register, enables IRQ

Use

This call is issued by ADFS to enable ST506 hard disc expansion cards to intercept ADFS and use their own hardware rather than the hardware built into the machine. The expansion card should claim the service call, updating the passed registers to the values for its own hardware.

Service_ADFSPoduleIDE (Service Call &10801)

Issued by ADFS to locate an IDE expansion card

On entry

R1 = &10801 (reason code)
R2 = address of current IDE hard disc controller
R3 = address of IRQ status register for current hard disc controller
R4 = mask which, when ANDed with IRQ status register, gives non-zero value if
 IRQs are enabled
R5 = address of IRQ mask register for current hard disc controller
R6 = mask which, when ORd with IRQ mask register, enables IRQ
R7= address of data read routine for current hard disc controller (0 for default)
R8 = address of data write routine for current hard disc controller (0 for default)

On exit

All registers preserved to pass on, else:

R1 = 0 to claim
R2 = address of new IDE hard disc controller
R3 = address of IRQ status register for new hard disc controller
R4 = mask which, when ANDed with IRQ status register, gives non-zero value if
 IRQs are enabled
R5 = address of IRQ mask register for new hard disc controller
R6 = mask which, when ORd with IRQ mask register, enables IRQ
R7= address of data read routine for new hard disc controller (0 for default)
R8 = address of data write routine for new hard disc controller (0 for default)

Use

This call is issued by ADFS to enable IDE hard disc expansion cards to intercept ADFS and use their own hardware rather than the hardware built into the machine. The expansion card should claim the service call, updating the passed registers to the values for its own hardware.

Service_ADFSPoduleIDEDying (Service Call &10802)

IDE expansion card dying

On entry

R1 = &10802 (reason code)

On exit

All registers preserved

Use

This call is issued by an IDE expansion card module to warn ADFS of its imminent demise.

SWI calls

Podule_ReadID (SWI &40280)

Reads an expansion card or extension ROM's identity byte

On entry

R3 = ROM section (see page 4-133)

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. It also resets the Loader – if one is present, and has been loaded.

Related SWIs

Podule_ReadHeader (page 4-140)

Related vectors

None

Podule_ReadHeader (SWI &40281)

Reads an expansion card or extension ROM's header

On entry

R2 = pointer to buffer of 8 or 16 bytes

R3 = ROM section (see page 4-133)

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. This call also resets the Loader – if one is present, and has been loaded.

If you do not know whether the card has Interrupt Status Pointers, you should use a 16 byte buffer. Extension ROMs always have Interrupt Status Pointers (although they're always zero), so you should always use a 16 byte buffer for them.

Related SWIs

Podule_ReadID (page 4-139)

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 = ROM section (see page 4-133)

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 a copy of the module's name if the chunk is a relocatable module
(ie if R2 = &81), 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 a copy of 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 correct ROM section. You should then repeatedly call this SWI until R0 is set to 0 on exit.

RISC OS 2 automatically does this on a reset for all expansion cards; if there is a Loader it will be transparently loaded, and any chunks in the code space will also be enumerated. Later versions of RISC OS use `Podule_EnumerateChunksWithInfo`.

Related SWIs

`Podule_ReadChunk` (page 4-144), `Podule_EnumerateChunksWithInfo` (page 4-157)

Related vectors

None

Podule_ReadChunk (SWI &40283)

Reads a chunk from an expansion card or extension ROM

On entry

R0 = chunk number
R2 = pointer to buffer (assumed large enough)
R3 = ROM section (see page 4-133)

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 (see page 4-142) to find the size of the chunk.

Related SWIs

Podule_EnumerateChunks (page 4-142)

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 the kernel does this automatically on a reset when it enumerates all expansion cards' chunks.

This command returns an error for extension ROMs, because they have neither code space nor a Loader.

Related SWIs

Podule_WriteBytes (page 4-147)

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 the kernel does this automatically on a reset when it enumerates all expansion cards' chunks.

This command returns an error for extension ROMs, because they have neither code space nor a Loader.

Related SWIs

Podule_ReadBytes (page 4-145)

Related vectors

None

Podule_CallLoader (SWI &40286)

Calls an expansion card's Loader

On entry

R0 - R2 = user data
R3 = expansion card slot number

On exit

R0 - R2 = user data

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

Depends on Loader

Use

This call enters an expansion card's Loader at offset 12. Registers R0 - R2 can be used to pass data.

The action the Loader takes will vary from card to card, and you should consult your card's documentation for further details.

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.

In some hardware designs it may be important to share hardware between the Loader and the driver. You can do so by using this call to call the Loader, which can do hardware accesses for the driver and maintain its own state. For example, if your

hardware has a 7 bit page register and a 1 bit output port shared within a single 8 bit latch, the Loader could maintain a flag for the state of the port, and write that bit correctly whenever it writes to the page register.

This command returns an error for extension ROMs, because they have neither code space nor a Loader.

Related SWIs

None

Related vectors

None

Podule_RawRead (SWI &40287)

Reads bytes directly within an expansion card or extension ROM's address space

On entry

R0 = offset from base of a podule's address space (0...&FFF)
R1 = number of bytes to read
R2 = pointer to buffer
R3 = ROM section (see page 4-133)

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 or extension ROM's address space. It is typically used to read from the registers of hardware devices on an expansion card, or to read successive bytes from an extension ROM.

You should use Podule_ReadBytes (page 4-145) to read from within an expansion card's code space.

Related SWIs

Podule_RawWrite (page 4-153)

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...&FFF)

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.

You should use Podule_WriteBytes (see page 4-147) to write within an expansion card's code space.

Obviously you cannot write to an extension ROM. You must not use this call to try to write to the ROM area; if you do so, you risk reprogramming the memory and video controllers.

Related SWIs

Podule_RawRead (page 4-151)

Related vectors

None

Podule_HardwareAddress (SWI &40289)

Returns an expansion card or extension ROM's base address, and the address of an expansion card's CMOS RAM

On entry

R3 = ROM section (see page 4-133), or base address of expansion card/extension ROM

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 or extension ROM's combined hardware address:

Bits	Meaning
0 - 11	base address of CMOS RAM – expansion cards only (4 bytes)
12 - 25	bits 12 - 25 of base address of expansion card/extension ROM
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.

In practice there is little point in finding the combined hardware address of an extension ROM. The base address of the extension ROM is of little use, as the width of the ROM can vary; and extension ROMs do not have CMOS RAM reserved for them.

Related SWIs

OS_Byte 161 (page 1-371), OS_Byte 162 (page 1-373),
Podule_HardwareAddresses (page 4-159)

Related vectors

None

Podule_EnumerateChunksWithInfo (SWI &4028A)

Reads information about a chunk from the Chunk Directory

On entry

R0 = chunk number (zero to start)
R3 = ROM section (see page 4-133)

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 a copy of the module's name if the chunk is a relocatable module, else preserved
R5 = pointer to a copy of the module's help string if the chunk is a relocatable module, else preserved
R6 = address of module if the chunk is a directly executable relocatable module, or 0 if the chunk is a non-directly-executable 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 pointers to copies of its name and its help string, and its address if it is executable. These are 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 correct ROM section. 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 expansion cards; if there is a Loader it will be transparently loaded, and any chunks in the code space will also be enumerated.

This call is not available in RISC OS 2, which uses `Podule_EnumerateChunks` instead.

Related SWIs

`Podule_EnumerateChunks` (page 4-142), `Podule_ReadChunk` (page 4-144)

Related vectors

None

Podule_HardwareAddresses (SWI &4028B)

Returns an expansion card or extension ROM's base address, and the address of an expansion card's CMOS RAM

On entry

R3 = ROM section (see page 4-133)

On exit

R0 = base address of expansion card/extension ROM

R1 = 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 or extension ROM's base address, and its combined hardware address:

Bits	Meaning
0 - 11	base address of CMOS RAM – expansion cards only (4 bytes)
12 - 25	bits 12 - 25 of base address of expansion card/extension ROM
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.

In practice there is little point in finding the combined hardware address of an extension ROM. The base address of the extension ROM is of little use, as the width of the ROM can vary; and extension ROMs do not have CMOS RAM reserved for them.

This call is not available in RISC OS 2.

Related SWIs

OS_Byte 161 (page 1-371), OS_Byte 162 (page 1-373),
Podule_HardwareAddress (page 4-155)

Related vectors

None

Podule_ReturnNumber (SWI &4028C)

Returns the number of expansion cards and extension ROMs

On entry

—

On exit

R0 = number of expansion cards

R1 = number of extension ROMs

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 the number of expansion cards and extension ROMs. The number of expansion cards returned is currently always 4, but you must be prepared to handle any other value, including 0.

This call is used by the *Podules command.

This call is not available in RISC OS 2.

Related SWIs

None

Related vectors

None

* Commands

*PoduleLoad

Copies a file into an expansion card's RAM

Syntax

```
*PoduleLoad expansion_card_number filename [offset]
```

Parameters

<i>expansion_card_number</i>	the expansion card's number, as given by *Podules
<i>filename</i>	a valid pathname, specifying a file
<i>offset</i>	offset (in hexadecimal by default) into the Code Space

Use

*PoduleLoad copies the contents of a file into an installed expansion card's RAM, starting at the specified offset. If no offset is given, then a default value of 0 is used.

Example

```
*PoduleLoad 1 $.Midi.Data 100
```

Related commands

*Podules, *PoduleSave

Related SWIs

Podule_WriteBytes (page 4-147)

Related vectors

None

*Podules

Displays a list of the installed expansion cards and extension ROMs

Syntax

*Podules

Parameters

None

Use

*Podules displays a list of the installed expansion cards and extension ROMs, using the description that each one holds internally. Some expansion cards and/or extension ROMs – such as one that is still being designed – will not have a description; in this case, an identification number is displayed.

This command still refers to expansion cards as podules, to maintain compatibility with earlier operating systems. This command does not show extension ROMs under RISC OS 2.

Example

*Podules

Podule 0: Midi and BBC I/O podule

Podule 1: Simple podule &8

Podule 2: No installed podule

Podule 3: No installed podule

Related commands

None

Related SWIs

Podule_EnumerateChunks (page 4-142)

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

<i>expansion_card_number</i>	the expansion card's number, as given by *Podules
<i>filename</i>	a valid pathname, specifying a file
<i>size</i>	in bytes
<i>offset</i>	offset (in hexadecimal by default) into the Code Space

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 200 100
```

Related commands

*Podules, *PoduleLoad

Related SWIs

Podule_ReadBytes (page 4-145)

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 > &.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$=MID$(Date$,5,2)+"-"+MID$(Date$,8,3)+"-"+MID$(Date$,14,2)
500 PROCAddString( &F2, Date$ )
530 REM PROCHeader( &F0, Z%+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%, S$ )

```

```

710 $$=$$+CHR$0
720 IF L% THEN PROCAddNormalString ELSE PROCAddPseudoString
730 ENDPROC
750 DEF PROCAddNormalString
760 IF Top%-Bot% < 10+LEN($$) THEN STOP
770 PROCHeader( T%, Top%-LEN($$)-Base%, LEN($$) )
780 Top%=Top%-LEN($$):PTR#H%=Top%:FOR I%=1 TO LEN($$)
790 BPUT#H%,ASC(MID$($$,I%,1)):NEXTI%:ENDPROC
810 DEF PROCAddPseudoString
820 IF Max%+Low%-Bot% < 9 THEN STOP
830 PROCHeader( T%, Top%-Base%, LEN($$) )
840 PTR#H%=Top%:FOR I%=1 TO LEN($$)
850 BPUT#H%,ASC(MID$($$,I%,1)):NEXTI%
860 Top%=Top%+LEN($$):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$;" not found.":=FALSE
980 S%=EXT#F%
990 IF L% THEN =FNAddNormalFile ELSE =FNAddPseudoFile
1010 DEF FNAddNormalFile
1020 E%=S%+9-(Top%-Bot%)
1030 IF E%>0 THEN PRINT"Oversize by ";E%;" bytes.":
      PROCChange:=FNAddPseudoFile
1040 PROCHeader( T%, Top%-S%-Base%, S% )
1050 Top%=Top%-S%:PTR#H%=Top%:FOR I%=1 TO S%
1060 BPUT#H%,BGET#F%:NEXTI%:CLOSE#F%:=TRUE
1080 DEF FNAddPseudoFile
1090 IF Max%+Low%-Bot% < 9 THEN =FALSE
1100 PROCHeader( T%, Top%-Base%, S% )
1110 PTR#H%=Top%
1120 FOR I%=1 TO S%:BPUT#H%,BGET#F%:NEXTI%
1130 Top%=Top%+S%:CLOSE#F%:=TRUE
1150 DEF PROCChange
1160 PRINT"Changing up. Wasting ";Top%-Bot%;" bytes."
1170 PTR#H%=Bot%:PROCByte(0):REM Terminate bottom directory
1180 Bot%=Max%:Top%=Max%+Low%:Base%=Max%:L%=FALSE
1190 REM In the pseudo area files grow upward from Top%
1200 ENDPROC
1220 DEF FNTtype( N$ )
1230 $Buffer%=N$:X%=Block%:Y%=X%/256:A%=5:X%!0=Buffer%
1240 B%=USR&FFDD:IF (B%AND255) <> 1 THEN PRINT"Not a file":=0
1250 V%=(Block%!3)AND&FFFFFF
1260 IFV%=&FFFFFFA THEN =&81
1270 IF((Block%!2AND&FFFF)=&8000)AND((Block%!6AND&FFFF)=&8000)THEN=&82
1280 IFV%=&FFFFFF9 THEN =&83
1290 =0

```


Introduction

The Acorn RISC machine has a general coprocessor interface. The first coprocessor available is one which performs floating point calculations to the IEEE standard. To ensure that programs using floating point arithmetic remain compatible with all Archimedes machines, a standard ARM floating point instruction set has been defined. This can be implemented invisibly to the customer program by one of several systems offering various speed performances at various costs. The current 'bundled' floating point system is the software-only floating point emulator module. Floating point instructions may be incorporated into any assembler text, provided they are called from user mode. These instructions are recognised by the Assembler and converted into the correct coprocessor instructions. However, these instructions are not supported by the assembler in 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), this chapter is structured differently from most 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.

Generally, programs do not need to know whether a coprocessor 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 coprocessor 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 formats described below.

There is also a *floating point status register* (FPSR) which, like the ARM's combined PC and PSR, holds all the necessary status and control information that an application is intended to be able to access. It holds *flags* which indicate various error conditions, such as overflow and division by zero. Each flag has a corresponding *trap enable bit*, which can be used to enable or disable a 'trap' associated with the error condition. Bits in the FPSR allow a client to distinguish between different implementations of the floating point system.

There may also be a *floating point control register* (FPCR); this is used to hold status and control information that an application is not intended to access. For example, there are privileged instructions to turn the floating point system on and off, to permit efficient context changes. Typically, hardware based systems have an FPCR, whereas software based ones do not.

Available systems

Floating point systems may be built from software only, hardware only, or some combination of software and hardware. The following terminology will be used to differentiate between the various ARM floating point systems already in use or planned:

System name	System components
Old FPE	Versions of the floating point emulator up to (but not including) 4.00
FPPC	Floating Point Protocol Converter (interface chip between ARM and WE32206), WE32206 (AT&T Math Acceleration Unit chip), and support code
FPE 400	Versions of the floating point emulator from 4.00 onwards
FPA	ARM Floating Point Accelerator chip, and support code

The results look the same to the programmer. However, if clients are aware of which system is in use, they may be able to extract better performance.

The old FPE has two different variants. Versions up to (but not including) 3.40 do not provide any hardware support, whereas versions 3.40 to 3.99 inclusive provide support for the FPPC hardware – if it is fitted. All versions of the FPE 400 provide support for the FPA hardware.

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’; in the event of a tie, this rounds to ‘nearest even’. If any of the others are 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. Specific instructions that work only with single precision operands may provide higher performance in some implementations, particularly the fully software based ones.

Floating point number formats

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 five formats (only four of which are visible at any one time, since P and EP are mutually exclusive):

IEEE Single Precision (S)



Figure 78.1 Single precision format

- If the exponent is 0 and the fraction is 0, the number represented is ± 0 .
- If the exponent is 0 and the fraction is non-zero, the number represented is $\pm 0.fraction \times 2^{-126}$.
- If the exponent is in the range 1 to 254, the number represented is $\pm 1.fraction \times 2^{exponent - 127}$.
- If the exponent is 255 and the fraction is 0, the number represented is $\pm \infty$.
- If the exponent is 255 and the fraction is non-zero, a NaN (not-a-number) is represented. If the most significant bit of the fraction is set, it is a non-trapping NaN; otherwise it is a trapping NaN.

IEEE Double Precision (D)

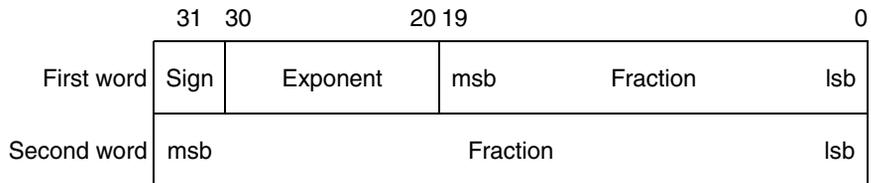


Figure 78.2 Double precision format

- If the exponent is 0 and the fraction is 0, the number represented is ± 0 .
- If the exponent is 0 and the fraction is non-zero, the number represented is $\pm 0.fraction \times 2^{-1022}$.
- If the exponent is in the range 1 to 2046, the number represented is $\pm 1.fraction \times 2^{exponent - 1023}$.
- If the exponent is 2047 and the fraction is 0, the number represented is $\pm \infty$.
- If the exponent is 2047 and the fraction is non-zero, a NaN (not-a-number) is represented. If the most significant bit of the fraction is set, it is a non-trapping NaN; otherwise it is a trapping NaN.

Double Extended Precision (E)

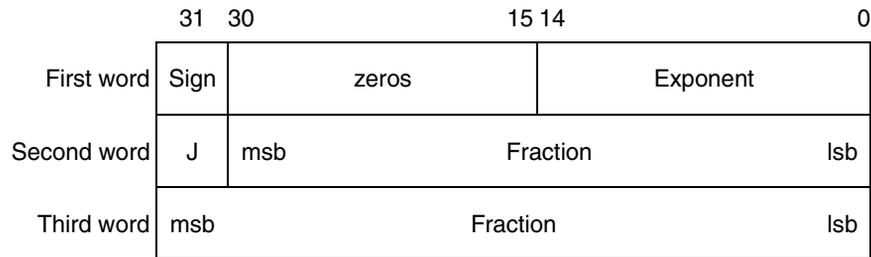


Figure 78.3 Double extended precision format

- If the exponent is 0, J is 0, and the fraction is 0, the number represented is ± 0 .
- If the exponent is 0, J is 0, and the fraction is non-zero, the number represented is $\pm 0.fraction \times 2^{-16382}$.
- If the exponent is in the range 0 to 32766, J is 1, and the fraction is non-zero, the number represented is $\pm 1.fraction \times 2^{exponent - 16383}$.
- If the exponent is 32767, J is 0, and the fraction is 0, the number represented is $\pm \infty$.
- If the exponent is 32767 and the fraction is non-zero, a NaN (not-a-number) is represented. If the most significant bit of the fraction is set, it is a non-trapping NaN; otherwise it is a trapping NaN.

Other values are illegal and shall not be used (ie the exponent is in the range 1 to 32766 and J is 0; or the exponent is 32767, J is 1, and the fraction is 0).

The FPPC system stores the sign bit in bit 15 of the first word, rather than in bit 31.

Storing a floating point register in 'E' format is guaranteed to maintain precision when loaded back by the same floating point system in this format. Note that in the past the layout of E format has varied between floating point systems, so software should not have been written to depend on it being readable by other floating point systems. For example, no software should have been written which saves E format data to disc, to have then been potentially loaded into another system. In particular, E format in the FPPC system varies from all other systems in its positioning of the sign bit. However, for the FPA and the FPE 400, the E format is now defined to be a particular form of IEEE Double Extended Precision and will not vary in future.

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

Figure 78.4 Packed decimal format

The sign nibble contains both the significand's sign (top bit) and the exponent's sign (next bit); the other two bits are zero.

d18 is the most significant digit of the significand d , and e3 of the exponent e . The significand has an assumed 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 and 3 digits respectively; d0, d1 and e3 may always be zero in a particular system. (By comparison, an S format number has 9 digits of significand and a maximum exponent of 53; a D format number has 17 digits in the significand and a maximum exponent of 340.)

The result is undefined if any of the packed digits is hexadecimal A - F, save for a representation of $\pm\infty$ or a NaN (see below).

- If the exponent's sign is 0, the exponent is 0, and the significand is 0, the number represented is ± 0 .
Zero will always be output as +0, but either +0 or -0 may be input.
- If the exponent is in the range 0 to 9999 and the significand is in the range 1 to 9.9999999999999999, the number represented is $\pm d \times 10^{\pm e}$.
- If the exponent is &FFFF (ie all the bits in e3 - e0 are set) and the significand is 0, the number represented is $\pm\infty$.
- If the exponent is &FFFF and d0 - d17 are non-zero, a NaN (not-a-number) is represented. If the most significant bit of d18 is set, it is a non-trapping NaN; otherwise it is a trapping NaN.

All other combinations are undefined.

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.

The FPSR consists of a system ID byte, an exception trap enable byte, a system control byte and a cumulative exception flags byte.

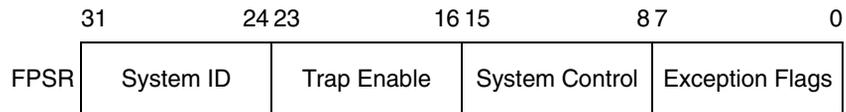


Figure 78.6 Floating point status register byte usage

System ID byte

The System ID byte allows a user or operating system to distinguish which floating point system is in use. The top bit (bit 31 of the FPSR) is set for **hardware** (ie fast) systems, and clear for **software** (ie slow) systems. Note that the System ID is read-only.

The following System IDs are currently defined:

System	System ID
Old FPE	&00
FPPC	&80
FPE 400	&01
FPA	&81

Exception Trap Enable Byte

Each bit of the exception trap enable byte corresponds to one type of floating point exception, which are described in the section entitled *Cumulative Exception Flags Byte* on page 4-178.

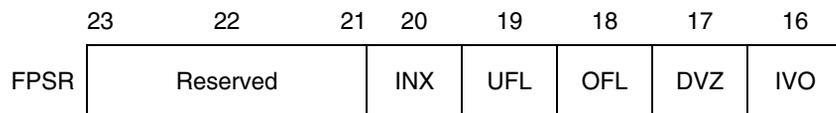


Figure 78.7 Exception trap enable byte

If a bit in the cumulative exception flags byte is set as a result of executing a floating point instruction, and the corresponding bit is also set in the exception trap enable byte, then that exception trap will be taken.

Currently, the reserved bits shall be written as zeros and will return 0 when read.

System Control Byte

These control bits determine which features of the floating point system are in use.

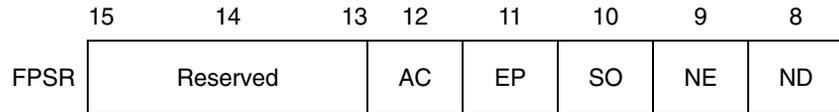


Figure 78.8 System control byte

By placing these control bits in the FPSR, their state will be preserved across context switches, allowing different processes to use different features if necessary. The following five control bits are defined for the FPA system and the FPE 400:

ND	No Denormalised numbers
NE	NaN Exception
SO	Select synchronous Operation of FPA
EP	Use Expanded Packed decimal format
AC	Use Alternative definition for C flag on compare operations

The old FPE and the FPPC system behave as if all these bits are clear.

Currently, the reserved bits shall be written as zeros and will return 0 when read. Note that all bits (including bits 8 - 12) are reserved on FPPC and early FPE systems.

ND – No denormalised numbers bit

If this bit is set, then the software will force all denormalised numbers to zero to prevent lengthy execution times when dealing with denormalised numbers. (Also known as abrupt underflow or flush to zero.) This mode is not IEEE compatible but may be required by some programs for performance reasons.

If this bit is clear, then denormalised numbers will be handled in the normal IEEE-conformant way.

NE – NaN exception bit

If this bit is set, then an attempt to store a signalling NaN that involves a change of format will cause an exception (for full IEEE compatibility).

If this bit is clear, then an attempt to store a signalling NaN that involves a change of format will not cause an exception (for compatibility with programs designed to work with the old FPE).

SO – Select synchronous operation of FPA

If this bit is set, then all floating point instructions will execute synchronously and ARM will be made to busy-wait until the instruction has completed. This will allow the precise address of an instruction causing an exception to be reported, but at the expense of increased execution time.

If this bit is clear, then that class of floating point instructions that can execute asynchronously to ARM will do so. Exceptions that occur as a result of these instructions may be raised some time after the instruction has started, by which time the ARM may have executed a number of instructions following the one that has failed. In such cases the address of the instruction that caused the exception will be imprecise.

The state of this bit is ignored by software-only implementations, which always operate synchronously.

EP – Use expanded packed decimal format

If this bit is set, then the expanded (four word) format will be used for Packed Decimal numbers. Use of this expanded format allows conversion from extended precision to packed decimal and back again to be carried out without loss of accuracy.

If this bit is clear, then the standard (three word) format is used for Packed Decimal numbers.

AC – Use alternative definition for C flag on compare operations

If this bit is set, the ARM C flag, after a compare, is interpreted as ‘Greater Than or Equal or Unordered’. This interpretation allows more of the IEEE predicates to be tested by means of single ARM conditional instructions than is possible using the original interpretation of the C flag (as shown below).

If this bit is clear, the ARM C flag, after a compare, is interpreted as ‘Greater Than or Equal’.

Cumulative Exception Flags Byte

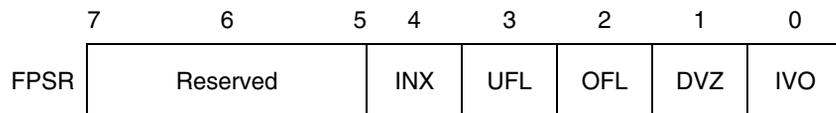


Figure 78.9 Cumulative exception flags byte

Whenever an exception condition arises, the appropriate cumulative exception flag in bits 0 to 4 will be set to 1. If the relevant trap enable bit is set, then an exception is also delivered to the user’s program in a manner specific to the operating system. (Note that

in the case of underflow, the state of the trap enable bit determines under which conditions the underflow flag will be set.) These flags can only be cleared by a WFS instruction.

Currently, the reserved bits shall be written as zeros and will return 0 when read.

IVO – invalid operation

The IVO flag is set when an operand is invalid for the operation to be performed. Invalid operations are:

- Any operation on a trapping NaN (not-a-number)
- Magnitude subtraction of infinities, eg $+\infty + -\infty$
- Multiplication of 0 by $\pm\infty$
- Division of 0/0 or ∞/∞
- $x \text{ REM } y$ where $x = \infty$ or $y = 0$
(REM is the ‘remainder after floating point division’ operator.)
- Square root of any number < 0 (but $\sqrt{(-0)} = -0$)
- Conversion to integer or decimal when overflow, ∞ or a NaN operand make it impossible

If overflow makes a conversion to integer impossible, then the largest positive or negative integer is produced (depending on the sign of the operand) and IVO is signalled

- Comparison with exceptions of Unordered operands
- ACS, ASN when argument’s absolute value is > 1
- SIN, COS, TAN when argument is $\pm\infty$
- LOG, LGN when argument is ≤ 0
- POW when first operand is < 0 and second operand is not an integer, or first operand is 0 and second operand is ≤ 0
- RPW when first operand is not an integer and second operand is < 0 , or first operand is ≤ 0 and second operand is 0.

DVZ – division by zero

The DVZ flag is set if the divisor is zero and the dividend a finite, non-zero number. A correctly signed infinity is returned if the trap is disabled.

The flag is also set for LOG(0) and for LGN(0). Negative infinity is returned if the trap is disabled.

OFL – overflow

The OFL flag is set whenever the destination format's largest number is exceeded in magnitude by what the rounded result would have been were the exponent range unbounded. As overflow is detected after rounding a result, whether overflow occurs or not after some operations depends on the rounding mode.

If the trap is disabled either a correctly signed infinity is returned, or the format's largest finite number. This depends on the rounding mode and floating point system used.

UFL – underflow

Two correlated events contribute to underflow:

- *Tininess* – the creation of a tiny non-zero result smaller in magnitude than the format's smallest normalised number.
- *Loss of accuracy* – a loss of accuracy due to denormalisation that **may** be greater than would be caused by rounding alone.

The UFL flag is set in different ways depending on the value of the UFL trap enable bit. If the trap is enabled, then the UFL flag is set when tininess is detected regardless of loss of accuracy. If the trap is disabled, then the UFL flag is set when both tininess and loss of accuracy are detected (in which case the INX flag is also set); otherwise a correctly signed zero is returned.

As underflow is detected after rounding a result, whether underflow occurs or not after some operations depends on the rounding mode.

INX – inexact

The INX flag 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, or underflow has occurred while the UFL trap was disabled. OFL or UFL traps take precedence over INX.

The INX flag is also set when computing SIN or COS, with the exceptions of SIN(0) and COS(1).

The old FPE and the FPPC system may differ in their handling of the INX flag. Because of this inconsistency we recommend that you do not enable the INX trap.

Floating Point Control Register

The Floating Point Control register (FPCR) may only be present in some implementations: it is there to control the hardware in an implementation-specific manner, for example to disable the floating point system. The user mode of the ARM is not permitted to use this register (since the right is reserved to alter it between implementations) and the WFC and RFC instructions will trap if tried in user mode.

You are unlikely to need to access the FPCR; this information is principally given for completeness.

The FPPC system

The FPCR bit allocation in the FPPC system is as shown below:

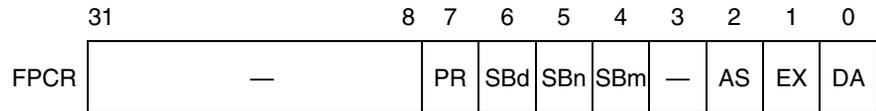


Figure 78.10 FPCR bit allocation in the FPPC system

Bit		Meaning
31-8		Reserved – always read as zero
7	PR	Last RMF instruction produced a partial remainder
6	SBd	Use Supervisor Register Bank ‘d’
5	SBn	Use Supervisor Register Bank ‘n’
4	SBm	Use Supervisor Register Bank ‘m’
3		Reserved – always read as zero
2	AS	Last WE32206 exception was asynchronous
1	EX	Floating point exception has occurred
0	DA	Disable

Reserved bits are ignored during write operations (but should be zero for future compatibility.) The reserved bits will return zero when read.

The FPA system

In the FPA, the FPCR will also be used to return status information required by the support code when an instruction is bounced. You should not alter the register unless you really know what you’re doing. Note that the register will be read sensitive; **even reading the register may change its value, with disastrous consequences.**

The FPCR bit allocation in the FPA system is **provisionally** as follows:

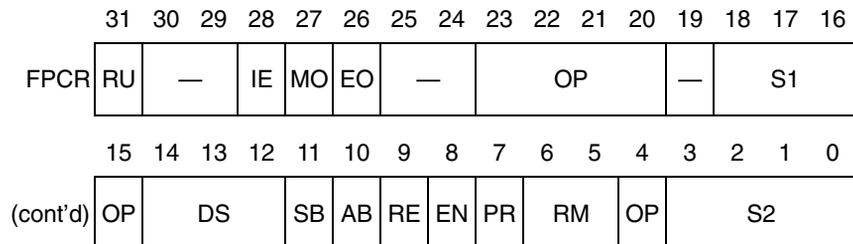


Figure 78.11 FPCR bit allocation in the FPA system

Bit		Meaning
31	RU	Rounded Up Bit
30		Reserved
29		Reserved
28	IE	Inexact bit
27	MO	Mantissa overflow
26	EO	Exponent overflow
25, 24		Reserved
23-20	OP	AU operation code
19	PR	AU precision
18-16	S1	AU source register 1
15	OP	AU operation code
14-12	DS	AU destination register
11	SB	Synchronous bounce: decode (R14) to get opcode
10	AB	Asynchronous bounce: opcode supplied in rest of word
9	RE	Rounding Exception: Asynchronous bounce occurred during rounding stage and destination register was written
8	EN	Enable FPA (default is off)
7	PR	AU precision
6, 5	RM	AU rounding mode
4	OP	AU operation code
3-0	S2	AU source register 2 (bit 3 set denotes a constant)

Note that the SB and AB bits are cleared on a read of the FPCR. Only the EN bit is writable. All other bits shall be set to zero on a write.

The instruction set

Floating point coprocessor data transfer

<i>op{condition}prec</i>	<i>Fd,addr</i>
<i>op</i>	is LDF for load, STF for store
<i>condition</i>	is one of the usual ARM conditions (see <i>Appendix A: ARM assembler</i> on page 4-361)
<i>prec</i>	is one of the usual floating point precisions (eg S for single, D for double, P for packed decimal: see the section entitled <i>Floating point number formats</i> on page 4-171)
<i>addr</i>	is <i>[Rn]{,#offset}</i> or <i>[Rn,#offset]{!}</i> (<i>!</i> if present indicates that writeback is to take place.)
<i>Fd</i>	is a floating point register symbol (defined via the FN directive).

Load (LDF) or store (STF) the high precision value from or to memory, using one of the five 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. Thus other rounding methods may be used by having previously applied some suitable floating point data operation; this does not compromise the requirement of ‘rounding once only’, since the store operation introduces no additional rounding error.

The offset is in words from the address given by the ARM base register, and is in the range -1020 to $+1020$. In pre-indexed mode you must explicitly specify writeback to add the offset to the base register; but in post-indexed mode the assembler forces writeback for you, as without write back post-indexing is meaningless.

You should not use R15 as the base register if writeback will take place.

Examples:

LDFS	F0,[R0]	; load F0 from address held in R0
		; (single precision)
STFP	F1,[R2]	; store number held in F1 at R2
		; as a packed decimal number

Floating point literals

LDFS and LDFD can be given literal values instead of a register relative address, and the Assembler will automatically place the required value in the next available literal pool. In the case of LDFS a single precision value is placed, in the case of LDFD a double precision value is placed. Because the allowed offset range within a LDFS or LDFD instruction is less than that for a LDR instruction (–1020 to +1020 instead of –4095 to +4095), it may be necessary to code LTORG directives more frequently if floating point literals are being used than would otherwise be necessary.

Syntax: LDFx Fn, = *floating point number*

Floating point coprocessor multiple data transfer

The LFM and SFM multiple data transfer instructions are supported by the assemblers, but are not provided by the FPPC system, or by some versions of the old FPE:

- versions 2.80 - 2.84 do not support them
- versions 2.85 - 3.39 do support them
- version 3.40 – which is effectively a version of 2.80 that also provides FPPC hardware support – does not support these instructions.

Attempting to execute these instructions on systems that do not provide them will cause undefined instruction traps, so you should only use these instructions in software intended for machines you are confident are using an appropriate version of the old FPE, or the FPE 400, or the FPA system.

The LFM and SFM instructions allow between 1 and 4 floating point registers to be transferred from or to memory in a single operation; such a transfer otherwise requires several LDF or STF operations. The multiple transfers are therefore useful for efficient stacking on procedure entry/exit and context switching. These new instructions are the preferred way to preserve exactly register contents within a program.

The values transferred to memory by SFM occupy three words for each register, but the data format used is not defined, and may vary between floating point systems. The only legal operation that can be performed on this data is to load it back into floating point registers using the LFM instruction. The data stored in memory by an SFM instruction should not be used or modified by any user process.

The registers transferred by a LFM or SFM instruction are specified by a base floating point register and the number of registers to be transferred. This means that a register set transferred has to have adjacent register numbers, unlike the unconstrained set of ARM registers that can be loaded or saved using LDM and STM. Floating point registers are transferred in ascending order, register numbers wrapping round from 7 to 0: eg transferring three registers with F6 as the base register results in registers F6, F7 then F0 being transferred.

The assembler supports two alternative forms of syntax, intended for general use or just stack manipulation:

op{*condition*} *Fd*,*count*,*addr*

op{*condition*}*stacktype* *Fd*,*count*,[*Rn*]{!}

op is LFM for load, SFM for store.

condition is one of the usual ARM conditions.

Fd is the base floating point register, specified as a floating point register symbol (defined via the FN directive).

count is an integer from 1 to 4 specifying the number of registers to be transferred.

addr is [*Rn*]{,*#offset*} or [*Rn*,*#offset*]{!}
({!} if present indicates that writeback is to take place).

stacktype is FD or EA, standing for Full Descending or Empty Ascending, the meanings as for LDM and STM.

The offset (only relevant for the first, general, syntax above) is in words from the address given by the ARM base register, and is in the range -1020 to +1020. In pre-indexed mode you must explicitly specify writeback to add the offset to the base register; but in post-indexed mode the assembler forces writeback for you, as without write back post-indexing is meaningless.

You should not use R15 as the base register if writeback will take place.

Examples:

SFMNE	F6,4,[R0]	;if NE is true, transfer F6, F7, ;F0 and F1 to the address ;contained in R0
LFMFD	F4,2,[R13]!	;load F4 and F5 from FD stack -
LFM	F4,2,[R13],#24	;equivalent to same instruction ;in general syntax

Floating point coprocessor register transfer

<i>FLT</i> { <i>condition</i> } <i>prec</i> { <i>round</i> }	<i>Fn</i> , <i>Rd</i>
<i>FLT</i> { <i>condition</i> } <i>prec</i> { <i>round</i> }	<i>Fn</i> ,# <i>value</i>
<i>FIX</i> { <i>condition</i> } <i>{round</i> }	<i>Rd</i> , <i>Fn</i>
<i>WFS</i> { <i>condition</i> }	<i>Rd</i>
<i>RFS</i> { <i>condition</i> }	<i>Rd</i>
<i>WFC</i> { <i>condition</i> }	<i>Rd</i>
<i>RFC</i> { <i>condition</i> }	<i>Rd</i>

{round} is the optional rounding mode: P, M or Z; see below.

Rd is an ARM register symbol.

Fn is a floating point register symbol.

The value may be of the following: 0, 1, 2, 3, 4, 5, 10, 0.5. Note that these values must be written precisely as shown above, for instance ‘0.5’ is correct but ‘.5’ is not.

<i>FLT</i>	Integer to Floating Point	<i>Fn</i> := <i>Rd</i>	
<i>FIX</i>	Floating point to integer	<i>Rd</i> := <i>Fm</i>	
<i>WFS</i>	Write Floating Point Status	<i>FPSR</i> := <i>Rd</i>	
<i>RFS</i>	Read Floating Point Status	<i>Rd</i> := <i>FPSR</i>	
<i>WFC</i>	Write Floating Point Control	<i>FPC</i> := <i>R</i>	Supervisor Only
<i>RFC</i>	Read Floating Point Control	<i>Rd</i> := <i>FPC</i>	Supervisor Only

The rounding modes are:

Mode	Letter
Nearest	(no letter required)
Plus infinity	P
Minus infinity	M
Zero	Z

Floating point coprocessor data operations

The formats of these instructions are:

<i>binop</i> { <i>condition</i> } <i>prec</i> { <i>round</i> }	<i>Fd</i> , <i>Fn</i> , <i>Fm</i>
<i>binop</i> { <i>condition</i> } <i>prec</i> { <i>round</i> }	<i>Fd</i> , <i>Fn</i> , # <i>value</i>
<i>unop</i> { <i>condition</i> } <i>prec</i> { <i>round</i> }	<i>Fd</i> , <i>Fm</i>
<i>unop</i> { <i>condition</i> } <i>prec</i> { <i>round</i> }	<i>Fd</i> , # <i>value</i>

binop is one of the binary operations listed below

unop is one of the unary operations listed below

Fd is the FPU destination register

Fn is the FPU source register (binops only)

F_m is the FPU source register
 $\#value$ is a constant, as an alternative to F_m . It must be 0, 1, 2, 3, 4, 5, 10 or 0.5, as above.

The binops are:

ADF	Add	$F_d := F_n + F_m$
MUF	Multiply	$F_d := F_n \times F_m$
SUF	Subtract	$F_d := F_n - F_m$
RSF	Reverse Subtract	$F_d := F_m - F_n$
DVF	Divide	$F_d := F_n / F_m$
RDF	Reverse Divide	$F_d := F_m / F_n$
POW	Power	$F_d := F_n$ to the power of F_m
RPW	Reverse Power	$F_d := F_m$ to the power of F_n
RMF	Remainder	$F_d :=$ remainder of F_n / F_m $(F_d := F_n - \text{integer value of } (F_n / F_m) \times F_m)$
FML	Fast Multiply	$F_d := F_n \times F_m$
FDV	Fast Divide	$F_d := F_n / F_m$
FRD	Fast Reverse Divide	$F_d := F_m / F_n$
POL	Polar angle	$F_d :=$ polar angle of F_n, F_m

The unops are:

MVF	Move	$F_d := F_m$
MNF	Move Negated	$F_d := -F_m$
ABS	Absolute value	$F_d := \text{ABS}(F_m)$
RND	Round to integral value	$F_d :=$ integer value of F_m
SQT	Square root	$F_d :=$ square root of F_m
LOG	Logarithm to base 10	$F_d := \log F_m$
LGN	Logarithm to base e	$F_d := \ln F_m$
EXP	Exponent	$F_d := e$ to the power of F_m
SIN	Sine	$F_d :=$ sine of F_m
COS	Cosine	$F_d :=$ cosine of F_m
TAN	Tangent	$F_d :=$ tangent of F_m
ASN	Arc Sine	$F_d :=$ arcsine of F_m
ACS	Arc Cosine	$F_d :=$ arccosine of F_m
ATN	Arc Tangent	$F_d :=$ arctangent of F_m
URD	Unnormalised Round	$F_d :=$ integer value of F_m (may be abnormal)
NRM	Normalise	$F_d :=$ normalised form of F_m

Note that wherever F_m is mentioned, one of the floating point constants 0, 1, 2, 3, 4, 5, 10, or 0.5 can be used instead.

FML, FRD and FDV are only defined to work with single precision operands. These ‘fast’ instructions are likely to be faster than the equivalent MUF, DVF and RDF instructions, but this is not necessarily so for any particular implementation.

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 URD and NRM operations are only supported by the FPA and the FPE 400.

Floating point coprocessor status transfer

op{condition}prec{round} Fm, Fn

op is one of the following:

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

{condition} an ARM condition.

prec a precision letter

{round} an optional rounding mode: P, M or Z

Fm A floating point register symbol.

Fn A floating point register symbol.

Compares are provided with and without the exception that could arise if the numbers are unordered (ie one or both of them is not-a-number). To comply with IEEE 754, the CMF instruction should be used to test for equality (ie when a BEQ or BNE is used afterwards) or to test for unorderedness (in the V flag). The CMFE instruction should be used for all other tests (BGT, BGE, BLT, BLE afterwards).

When the AC bit in the FPSR is clear, the ARM flags N, Z, C, V refer to the following after compares:

N	Less than	ie Fn less than Fm (or –Fm)
Z	Equal	
C	Greater than or equal	ie Fn greater than or equal to Fm (or –Fm)
V	Unordered	

Note that when two numbers are not equal, N and C are not necessarily opposites. If the result is unordered they will both be clear.

When the AC bit in the FPSR is set, the ARM flags N, Z, C, V refer to the following after compares:

N	Less than
Z	Equal
C	Greater than or equal or unordered
V	Unordered

In this case, N and C are necessarily opposites.

Finding out more...

Further details of the floating point instructions (such as the format of the bitfields within the instruction) can be found in the *Acorn RISC Machine family Data Manual*. VLSI Technology Inc. (1990) Prentice-Hall, Englewood Cliffs, NJ, USA: ISBN 0-13-781618-9 and in the *Acorn Assembler Release 2* manual.

SWI Calls

FPEmulator_Version (SWI &40480)

Returns the version number of the floating point emulator

On entry

—

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

79 ARM3 Support

Introduction and Overview

The ARM3 Support module provides commands to control the use of the ARM3 processor's cache, where one is fitted to a machine. The module will immediately kill itself if you try to run it on a machine that only has an ARM2 processor fitted.

Summary of facilities

Two * Commands are provided: one to configure whether or not the cache is enabled at a power-on or reset, and the other to independently turn the cache on or off.

There is also a SWI to turn the cache on or off. A further SWI forces the cache to be flushed. Finally, there is also a set of SWIs that control how various areas of memory interact with the cache.

The default setup is such that all RISC OS programs should run unchanged with the ARM3's cache enabled. Consequently, you are unlikely to need to use the SWIs (beyond, possibly, turning the cache on or off).

Notes

A few poorly-written programs may not work correctly with ARM3 processors, because they make assumptions about processor timing or clock rates.

This module is not available in RISC OS 2.00 (ie was introduced in RISC OS 2.01).

Finding out more

For more details of the ARM3 processor, see the *Acorn RISC Machine family Data Manual*. VLSI Technology Inc. (1990) Prentice-Hall, Englewood Cliffs, NJ, USA: ISBN 0-13-781618-9.

SWI Calls

Cache_Control (SWI &280)

Turns the cache on or off

On entry

R0 = XOR mask

R1 = AND mask

On exit

R0 = old state (0 \Rightarrow cacheing was disabled, 1 \Rightarrow cacheing was enabled)

Interrupts

Interrupts are disabled

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call turns the cache on or off. Bit 0 of the ARM3's control register 2 is altered by being masked with R1 and then exclusive ORd with R0: ie new value = ((old value AND R1) XOR R0). Bit 1 of the control register is also set, so the ARM 3 does **not** separately cache accesses to the same address for user and non-user modes. (To do so would degrade cache performance, and potentially cause cache inconsistency). Other bits of the control register are set to zero.

Related SWIs

None

Related vectors

None

Cache_Cacheable (SWI &281)

Controls which areas of memory may be cached

On entry

R0 = XOR mask

R1 = AND mask

On exit

R0 = old value (bit n set \Rightarrow 2MBytes starting at $n \times 2$ MBytes are cacheable)

Interrupts

Interrupts are disabled

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call controls which areas of memory may be cached (ie are *cacheable*). The ARM3's control register 3 is altered by being masked with R1 and then exclusive ORd with R0: ie new value = ((old value AND R1) XOR R0). If bit n of the control register is set, the 2MBytes starting at $n \times 2$ MBytes are cacheable.

The default value stored is &FC007CFF, so ROM and logical non-screen RAM are cacheable, but I/O space, physical memory, the RAM disc and logical screen memory are not.

Related SWIs

Cache_Updateable (page 4-196), Cache_Disruptive (page 4-198)

Related vectors

None

Cache_Updateable (SWI &282)

Controls which areas of memory will be automatically updated in the cache

On entry

R0 = XOR mask

R1 = AND mask

On exit

R0 = old value (bit n set \Rightarrow 2MBytes starting at $n \times 2$ MBytes are updateable)

Interrupts

Interrupts are disabled

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call controls which areas of memory will be automatically updated in the cache when the processor writes to that area (ie are *updateable*). The ARM3's control register 4 is altered by being masked with R1 and then exclusive ORd with R0: ie new value = ((old value AND R1) XOR R0). If bit n of the control register is set, the 2MBytes starting at $n \times 2$ MBytes are updateable.

The default value stored is &00007FFF, so logical non-screen RAM is updateable, but ROM/CAM/DAG, I/O space, physical memory and logical screen memory are not.

Related SWIs

Cache_Cacheable (page 4-194), Cache_Disruptive (page 4-198)

Related vectors

None

Cache_Disruptive (SWI &283)

Controls which areas of memory cause automatic flushing of the cache on a write

On entry

R0 = XOR mask

R1 = AND mask

On exit

R0 = old value (bit n set \Rightarrow 2MBytes starting at $n \times 2$ MBytes are disruptive)

Interrupts

Interrupts are disabled

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call controls which areas of memory cause automatic flushing of the cache when the processor writes to that area (ie are *disruptive*). The ARM3's control register 5 is altered by being masked with R1 and then exclusive ORd with R0: ie new value = ((old value AND R1) XOR R0). If bit n of the control register is set, the 2MBytes starting at $n \times 2$ MBytes are disruptive.

The default value stored is &F0000000, so the CAM map is disruptive, but ROM/DAG, I/O space, physical memory and logical memory are not. This causes automatic flushing whenever MEMC's page mapping is altered, which allows programs written for the ARM2 (including RISC OS itself) to run unaltered, but at the expense of unnecessary flushing on page swaps.

Related SWIs

Cache_Cacheable (page 4-194), Cache_Updateable (page 4-196)

Related vectors

None

Cache_Flush (SWI &284)

Flushes the cache

On entry

—

On exit

—

Interrupts

Interrupts are disabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This call flushes the cache by writing to the ARM3's control register 1.

Related SWIs

None

Related vectors

None

* Commands

*Cache

Turns the cache on or off, or gives the cache's current state

Syntax

*Cache [On|Off]

Parameters

On or Off

Use

*Cache turns the cache on or off. With no parameter, it gives the cache's current state.

Example

*Cache Off

Related commands

*Configure Cache

Related SWIs

Cache_Control (page 4-192)

Related vectors

None

*Configure Cache

Sets the configured cache state to be on or off

Syntax

*Configure Cache On|Off

Parameters

On or Off

Use

*Configure Cache sets the configured cache state to be on or off.

Example

*Configure Cache On

Related commands

*Cache

Related SWIs

Cache_Control (page 4-192)

Related vectors

None

Application Note

Games writers may wish to disable the ARM3 cache so that ARM3 based machines run at a similar speed to older ARM2 based machines. You must ensure that your code only tries to call ARM3Support SWIs and * Commands – such as *Cache Off – if the module is present. A simple way to do so is to call the error-returning form of an ARM3Support SWI, and see if an error is returned. For example:

```
SYS "XCache_Control",0,-1 TO R0;flags  
IF (flags AND 1) THEN arm3=FALSE ELSE arm3=TRUE  
IF arm3 THEN *Cache Off
```

80 The Portable module

Introduction

This module provides support for portable machines. The SWIs listed are not normally intended to be issued from user programs, they will normally be issued by other modules in the system.

Technical details

Colour to grey-scale mapping

The Portable module has to convert the users RGB palette settings into a grey-scale value in the range 0 to 14 (since the LCD panel only supports 15 unique grey levels). It does this using the following algorithm:

$$\text{Luminance} = (4 \times \text{Green}) + (2 \times \text{Red}) + \text{Blue}$$

Red, Green and Blue are in the range 0 to 255, so the luminance is in the range 0 to 1785 (255×7). It is then mapped down onto the range 0 to 14 using the following table:

Luminance	Grey level	Palette values for R, G and B
0 - 118	0	&00
119 - 237	1	&12
238 - 356	2	&24
357 - 475	3	&37
476 - 594	4	&49
595 - 713	5	&5B
714 - 832	6	&6D
833 - 952	7	&7F
953 - 1071	8	&92
1072 - 1190	9	&A4
1191 - 1309	10	&B6
1310 - 1428	11	&C8
1429 - 1547	12	&DB
1548 - 1666	13	&ED
1667 - 1785	14	&FF

The mapping table above is provided for information only, and may be subject to change in later versions of the Portable module.

In 256 colour modes the colour mapping is partly determined by the hardware, since the top 4 bits of the pixel value go directly to particular bits of the three guns, and the LCD ASIC only takes input from VIDC's red output. Thus the grey level will not in general map correctly from the luminance of the RGB value which would normally be output.

Service calls

Service_Portable (Service Call &8A)

Power down or up

On entry

R1 = reason code (&8A)

R2 = power up or down:

0 = power down

1 = power up

R3 = bit mask of which ports are being powered down (if R2 = 0)
(bit set \Rightarrow port is being powered down)

bit mask of which ports have been powered up (if R2 = 1)
(bit set \Rightarrow port has been powered up)

On exit

R1 = 0 if R3 = 0, else preserved to pass on

R2 preserved

R3 = bit mask of which ports may be powered down or up
(bit set \Rightarrow no objection to change of state)

Use

This call is issued before power is removed or after power is reapplied to the following:

Econet (bit 0)

serial buffer/oscillator (bit 3)

FDC oscillator (bit 14)

If a module wishes to prevent hardware being powered down, it should clear the appropriate bit or bits in R3. In addition, if the resulting value in R3 is now zero, the module should claim the service by setting R1 to zero. (This is to prevent the call being unnecessarily passed round the rest of the modules). Otherwise the service should be passed on by preserving R1.

This call should never be claimed.

SWI Calls

Portable_Speed (swi &42FC0)

Controls the processor speed

On entry

R0 = EOR mask
R1 = AND mask

On exit

R0 = old speed
R1 = new speed (0 \Rightarrow fast, 1 \Rightarrow slow)

Interrupts

Interrupt status is not defined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI controls the processor speed, which is reduced when the system is idle in order to save power.

The new speed is calculated as follows:

$$\text{new speed} = (\text{old speed AND R1}) \text{ EOR R0}$$

Speed settings currently supported are:

0 fast
1 slow

Related SWIs

Portable_Control (page 4-210)

Related vectors

None

Portable_Control (SWI &42FC1)

Controls various power control and miscellaneous bits

On entry

R0 = EOR mask

R1 = AND mask

On exit

R0 = old control

R1 = new control

Interrupts

Interrupt status is not defined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI controls various power control and miscellaneous bits in the portable machine.

The new control is calculated as follows:

$$\text{new control} = (\text{old control AND R1}) \text{ EOR R0}$$

The bits in control are as follows:

Bit	Meaning
0	Set ⇒ power to Econet enabled
1	Set ⇒ power to LCD display enabled
2	Set ⇒ power to external video display enabled
3	Set ⇒ power to serial buffer and oscillator enabled
4	Set ⇒ dual panel mode enabled
5, 6	Video clock control <ul style="list-style-type: none"> 0 ⇒ External clock input 1 ⇒ Crystal oscillator, divided by 2 2 ⇒ Crystal oscillator 3 ⇒ reserved, do not use
7	Set ⇒ invert video clock
8	Set ⇒ back-light enabled
9	Clear ⇒ 1 extra line on display Set ⇒ 2 extra lines on display
10	Clear ⇒ 1 DRAM used for dual panel Set ⇒ 2 DRAMs used for dual panel
11 - 13	Reserved
14	Set ⇒ power to FDC oscillator enabled
15	Reserved
16	Set ⇒ LCD palette set up for inverse video
17 - 31	Reserved

Reserved bits must not be modified, nor assumed to read any particular value.

Note that the 82C711 has one oscillator which is used by the serial subunit and by the floppy disc controller (FDC). Power to the oscillator is removed only if bits 3 and 14 are both clear.

On some computers the power to the oscillator cannot be removed because the same oscillator drives other parts of the system (eg IOEB).

If this call results in bits 0, 3 or 14 changing (ie power being removed or applied to the serial buffer/oscillator, Econet or FDC oscillator), then `Service_Portable` is issued (see page 4-207).

Related SWIs

`Portable_Speed` (page 4-208)

Related vectors

None

Portable_ReadBMUVariable (swi &42FC2)

Reads Battery Management Unit variables

On entry

R0 = BMU variable number

On exit

R0 preserved

R1 = value of variable

Interrupts

Interrupts enabled except if R0 = 10

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI reads Battery Management Unit variables.

The BMU variable numbers are:

Variable	Read/Write	Description																
0	R	version number and memory map of BMU microcode																
1	R	nominal battery capacity																
2	R	measured battery capacity																
3	R	used battery capacity																
4	R	usable battery capacity																
5	R	reserved																
6	R/W	charge estimate																
7	R	instantaneous voltage																
8	R	instantaneous current																
9	R	instantaneous temperature																
10	R	flags as follows: <table border="1" data-bbox="685 659 1085 911"> <thead> <tr> <th>Bit</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Set ⇒ lid is open</td> </tr> <tr> <td>2</td> <td>Set ⇒ threshold 2 reached</td> </tr> <tr> <td>3</td> <td>Set ⇒ threshold 1 reached</td> </tr> <tr> <td>4</td> <td>Set ⇒ charging system fault</td> </tr> <tr> <td>5</td> <td>Set ⇒ charge state is known</td> </tr> <tr> <td>6</td> <td>Set ⇒ battery present</td> </tr> <tr> <td>7</td> <td>Set ⇒ charger connected</td> </tr> </tbody> </table>	Bit	Meaning	1	Set ⇒ lid is open	2	Set ⇒ threshold 2 reached	3	Set ⇒ threshold 1 reached	4	Set ⇒ charging system fault	5	Set ⇒ charge state is known	6	Set ⇒ battery present	7	Set ⇒ charger connected
Bit	Meaning																	
1	Set ⇒ lid is open																	
2	Set ⇒ threshold 2 reached																	
3	Set ⇒ threshold 1 reached																	
4	Set ⇒ charging system fault																	
5	Set ⇒ charge state is known																	
6	Set ⇒ battery present																	
7	Set ⇒ charger connected																	
11	R	charge rate (bits 4 to 7)																

Reading any variable except the flags (variable 10) will enable IRQs (the flags are read from a soft copy).

Related SWIs

Portable_WriteBMUVariable (page 4-214)

Related vectors

None

Portable_WriteBMUVariable (SWI &42FC3)

Writes Battery Management Unit variables

On entry

R0 = BMU variable number

R1 = new value of variable

On exit

R0, R1 preserved

Interrupts

Interrupts status is not defined

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI writes Battery Management Unit variables.

The variable numbers are as for Portable_ReadBMUVariable on page 4-213. Variables not marked with a 'W' should not be written.

Related SWIs

Portable_ReadBMUVariable (page 4-212)

Related vectors

None

Portable_CommandBMU (SWI &42FC4)

Issues a command to the Battery Management Unit

On entry

R0 = reason code
1 = Remove power
2 = Reserved
3 = Reserved
4 = Set autostart (R1 = delay, in minutes, – 1; eg 0 ⇒ 1 minute delay)
Other registers hold reason-code-dependent parameters

On exit

All registers preserved

Interrupts

Interrupt status is not defined
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is not re-entrant

Use

This SWI issues a command to the Battery Management Unit. The values of variables after a command may not change immediately this command is issued.

Related SWIs

None

Related vectors

None



81 Joystick module

Introduction and Overview

The Joystick module provides a SWI interface for reading the state of a joystick. When the module initialises it tests for the existence of built-in joystick hardware and if it does not find any then it will not initialise. Third parties can replace this module to provide different hardware. It is recommended that any such modules have version numbers greater than 2.00 so that Acorn can upgrade its own module without preventing its replacement.

SWI Calls

Joystick_Read (SWI &43F40)

Returns the state of a joystick

On entry

R0 = joystick number

On exit

R0 = joystick state

Interrupts

Interrupt status is unaltered

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

Not defined

Use

This SWI is used to obtain the state of the requested joystick. The state is returned in the following format, which supports both digital and analogue devices:

Byte	Value
0	Signed Y value in the range -127 to 127. For a single switch joystick, -64 ⇒ Down, 0 ⇒ Rest, and 64 ⇒ Up.
1	Signed Y value in the range -127 to 127. For a single switch joystick, -64 ⇒ Left, 0 ⇒ Rest, and 64 ⇒ Right.
2	Switches (eg fire buttons) starting in bit 0; unimplemented switches return 0.
3	Reserved.

Applications which are only interested in state (up, down, left, right) should not simply test the bytes for positive, negative or zero. We recommend that the 'at rest' state should span a middle range, say from -32 to 32 , since analogue joysticks cannot be relied upon to produce 0 when at rest.

Related SWIs

None

Related vectors

None

Part 14 – Programmer's support

82 Debugger

Introduction

The debugger is a module that allows a 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 hexadecimal. A preceding & is optional; that is, unlike most of the rest of the system, the debugger uses hexadecimal as a default base rather than decimal.

*Quit should be used to return from the debugger to the previous environment after a breakpoint – see page 1-332.

Note that the breakpoints discussed here are separate from those caused by `OS_BreakPt`. See page 1-311 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’ 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.

From RISC OS 3 onwards this module supports ARM 3 instructions, and warns of certain unwise or invalid code sequences (see *Appendix B: Warnings on the use of ARM assembler* on page 4-383). Some of the output when disassembling has been changed for greater clarity than that provided by RISC OS 2.

SWI Calls

Debugger_Disassemble (SWI &40380)

Disassemble an instruction

On entry

R0 = instruction to disassemble
R1 = address to assume the instruction 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 from which to assume the instruction came, 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 excludes the zero-byte.

Related SWIs

None

Related vectors

None

*Commands

*BreakClr

Removes a breakpoint

Syntax

```
*BreakClr [addr reg]
```

Parameters

addr

hexadecimal address of breakpoint to clear

reg

register containing address of breakpoint to clear

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 or register value, putting the original contents back into that location. You can unset the last hit breakpoint with the command *BreakClr pc

If you give no parameter then you can remove all breakpoints – you will be prompted:

Clear all breakpoints [Y/N]?

Example

```
*BreakClr 816C
```

Related commands

*BreakSet, *BreakList

Related SWIs

None

Related vectors

None

***BreakList**

List all the breakpoints that are currently set

Syntax

**BreakList*

Parameters

None

Use

**BreakList* lists all the breakpoints that are currently set with **BreakSet*.

Example

```
*BreakList  
Address   Old Data  
0000816C EF00141C
```

Related commands

**BreakSet*

Related SWIs

None

Related vectors

None

*BreakSet

Sets a breakpoint

Syntax

`*BreakSet addrreg`

Parameters

addr

hexadecimal address of breakpoint to set

reg

register containing address of breakpoint to set

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

*BreakSet sets a breakpoint at the specified address or register value, so that when the code is executed and the instruction at that address is reached, execution will be halted.

When a breakpoint is set, the previous contents of the breakpoint address are replaced with a branch into the debugger code. This means that you may only set breakpoints in RAM. If you try to set a breakpoint in ROM, the error 'Bad breakpoint' is generated.

Example

```
*BreakSet 816C
```

Related commands

*BreakClr, *BreakList, *Continue

Related SWIs

None

Related vectors

None

*Continue

Resumes execution after a breakpoint

Syntax

*Continue

Parameters

None

Use

*Continue resumes execution after a breakpoint, using the saved state. If there is a breakpoint at the continuation position, then this prompt is given:

```
Continue from breakpoint set at &0000816C  
Execute out of line? [Y/N]?
```

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 (such as LDR R0,label, a B or BL instruction, or an ADR directive), you should not execute it out of line. Instead you should clear the breakpoint, and then re-issue the *Continue command. The instruction will then be executed in line, avoiding the wrong address being referenced.

Related commands

*BreakClr, *BreakList, *BreakSet

Related SWIs

None

Related vectors

None

*Debug

Enters the debugger

Syntax

*Debug

Parameters

None

Use

Debug enters the debugger. A prompt of Debug appears. Use Escape to return to the caller, or *Quit to exit to the caller's parent.

*Quit is documented on page 1-332.

Related commands

*Quit

Related SWIs

None

Related vectors

None

*InitStore

Fills user memory with a value

Syntax

*InitStore [*value*|*reg*]

Parameters

<i>value</i>	word with which to fill user memory
<i>reg</i>	register value with which to fill user memory 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 value or register value, or with the value &E6000010 (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.

RISC OS 2 used the value &E1000090 instead. This is no longer an illegal instruction for all versions of the ARM processor.

Example

```
*InitStore &381E6677
```

Related commands

None

Related SWIs

None

Related vectors

None

*Memory

Displays the values in memory

Syntax

```
*Memory [B] addr1|reg1
*Memory [B] addr1|reg1 [+|-]addr2|reg2
*Memory [B] addr1|reg1 +/-addr2|reg2 +addr3|reg3
```

Parameters

B	optionally display as bytes
<i>addr1</i> <i>reg1</i>	hexadecimal address, or register containing address for start of display
<i>addr2</i> <i>reg2</i>	hexadecimal offset, or register containing offset
<i>addr3</i> <i>reg3</i>	hexadecimal offset, or register containing offset

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 memory, in bytes if the optional B is given, or in words otherwise.

If only one address is given, 256 bytes are displayed starting from *addr1*. If two addresses are given, *addr2* specifies the end of the range to be displayed (as an absolute address or, if '+' or '-' is present, 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*).

Example

```
*Memory 1000 -200 +500
```

Display memory from &E00 to &12FF

Related commands

```
*MemoryA, *MemoryI
```

Related SWIs

None

Related vectors

None

*MemoryA

Displays and alters memory

Syntax

```
*MemoryA [B] addr|reg1 [value|reg2]
```

Parameters

<i>B</i>	optionally display as bytes
<i>addr reg1</i>	hexadecimal address, or register containing address for start of display
<i>value</i>	value to write into the specified location
<i>reg2</i>	register containing value to write into the specified location 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

*MemoryA displays and alters memory in bytes, if the optional B is given, or in words otherwise.

If you give no further parameters, interactive mode is entered. At each line, something similar to the following is printed:

```
*MemoryA 8000  
+ 00008000 : x•.. : 00008F78 : ANDEQ R8,R0,R8,ROR PC  
Enter new value :
```

or, for byte mode:

```
*MemoryA B 8001  
+ 00008001 : • : 8F :  
Enter new value :
```

The first character shows the direction in which Return steps ('+' for forwards, '-' for backwards). Next is the address of the word/byte being altered, then the character(s) in that word/byte, then the current hexadecimal value of the word/byte, and finally (for words only) the instruction at that address.

You may type any of the following at the prompt:

Return	to go to the 'next' location
-	to step backwards in memory
+	to step forwards in memory
<i>hex digits</i> Return	to alter a location and proceed
.	to exit.

As an alternative to using this command interactively, you can give the new data value on the line after the address.

Example

```
*MemoryA 87A0 12345678
```

Related commands

*Memory, *MemoryI

Related SWIs

None

Related vectors

None

*MemoryI

Disassembles memory into ARM instructions

Syntax

```
*MemoryI addr1\reg1
*MemoryI addr1\reg1 [+|-]addr2\reg2
*MemoryI addr1\reg1 +-addr2\reg2 +addr3\reg3
```

Parameters

<i>addr1</i> \reg1	hexadecimal address, or register containing address for start of display
<i>addr2</i> \reg2	hexadecimal offset, or register containing offset
<i>addr3</i> \reg3	hexadecimal offset, or register containing offset

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 into ARM instructions.

If only one address is given, 24 instructions are disassembled starting from *addr1*. If two addresses are given, *addr2* specifies the end of the range to be disassembled (as an absolute address or, if '+' or '-' is present, 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

No. Position Workspace Name

...

22 0184D684 018016B4 Debugger

...

Find address of Debugger

*memoryi 184D684 +24

0184D684 : : 00000000 : ANDEQ R0,R0,R0

0184D688 : \... : 0000005C : ANDEQ R0,R0,R12,ASR R0

0184D68C : (... : 00000128 : ANDEQ R0,R0,R8,LSR #2

0184D690 : : 00000104 : ANDEQ R0,R0,R4,LSL #2

0184D694 : (... : 00000028 : ANDEQ R0,R0,R8,LSR #32

0184D698 : >... : 0000003E : ANDEQ R0,R0,R14,LSR R0

0184D69C : h... : 00000168 : ANDEQ R0,R0,R8,ROR #2

0184D6A0 : ... : 00040380 : ANDEQ R0,R4,R0,LSL #7

0184D6A4 : ü... : 000005FC : MULEQ R0,R12,R5

← Offset of SWI handler is &5FC

*memoryi 184D684 +5FC +20

0184DC80 : .B-é : E92D4200 : STMDB R13!,{R9,R14}

0184DC84 : .À†ä : E49CC000 : LDR R12,[R12],#0

0184DC88 : ...ã : E33B0000 : TEQ R11,#0

0184DC8C : : 0A000005 : BEQ &0184DCA8

0184DC90 : ...â : E28F0004 : ADR R0,&0184DC9C

0184DC94 : _..ë : EB00075F : BL &0184FA18

0184DC98 : . /:è : E8BD8200 : LDMIA R13!,{R9,PC}

0184DC9C : : 0000010F : ANDEQ R0,R0,PC,LSL #2

Disassemble SWI handler

Related commands

*Memory, *MemoryA

Related SWIs

Debugger_Disassemble (page 4-225)

Related vectors

None

*ShowRegs

Displays the register contents for the saved state

Syntax

*ShowRegs

Parameters

None

Use

*ShowRegs displays the register contents for the saved state, which may be caught on one of the five following traps:

- undefined instruction
- address exception
- data abort
- prefetch abort
- break point.

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 &01804D2C) is:

```
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 = 2538DAF0 R14 = 24368000 R15 = 7629D100  
Mode USR flags set : nzcviif
```

Related commands

None

Related SWIs

None

**ShowRegs*

Related vectors

None

Introduction

The shared C library is a RISC OS relocatable module (called SharedCLibrary) which contains the whole of the ANSI C library. It is used by many programs written in C. Consequently, it saves both RAM space and disc space.

The shared C library is used by the RISC OS applications Edit, Paint, Draw and Configure.

Generally you will use the shared C library by linking your programs with the library stubs. However, you may also call it directly from assembly language by means of SWIs provided by the shared C library (you would normally only want to do this if you are implementing your own library stubs for your own language run-time system).

Overview

How to use the C library kernel

C library structure

The C library is organised into three layers:

- at the centre is the language-independent library kernel providing basic support services;
- at the next level is a C-specific layer providing compiler support functions;
- at the outermost level is the actual C library.

A full description of all the C library functions is given in the section entitled *C library functions* on page 4-288.

The library kernel

The library kernel is designed to allow run-time libraries for different languages to co-reside harmoniously, so that inter-language calling can be smooth. It provides the following facilities:

- a generic, status-returning, procedural interface to SWIs
- a procedural interface to commonly used SWIs, arithmetic functions and miscellaneous functions
- support for manipulating the IRQ state from a relocatable module
- support for allocating and freeing memory in the RMA area
- support for stack-limit checking and stack extension
- trap handling, error handling, event handling and escape handling.

A full description of all the library kernel functions is given in the section entitled *Library kernel functions* on page 4-275.

Interfacing a language run-time system to the Acorn library kernel

You can also write your own language Run-Time System to use the shared C library. For full details, see the section entitled *Interfacing a language run-time system to the Acorn library kernel* on page 4-242.

How the run-time stack is managed and extended

Management

The run-time stack consists of a doubly-linked list of stack chunks. Each stack chunk is allocated by the storage manager of the master language (in a C program allocating and freeing stack chunks is accomplished using `malloc()` and `free()`).

Stack extension

Two types of stack extension are provided:

- Pascal/Modula-2 style
- C-style

Calling other programs from C

The C library procedure system() provides the means whereby a program can pass a command to the host system's command line interpreter – in this case the RISC OS command line interpreter. For a full description, see the section entitled *Calling other programs from C* on page 4-243.

Storage management

The storage manager manages the heap in the most 'efficient' manner possible. A rudimentary understanding of it will help you make the best use of it; see the section entitled *Storage management* on page 4-243.

Handling host errors

Calls made to RISC OS via a kernel function return a specific value if an operating system error occurs. A call is provided to then find the associated error number and string. For full details, see the section entitled *Handling host errors* on page 4-253.

Technical details

The shared C library module implements a single SWI which is called by code in the library stubs when your program linked with the stubs starts running. That SWI call tells the stubs where the library is in the machine. This allows the vector of library entry points contained in the stubs to be patched up in order to point at the relevant entry points in the library module.

The stubs also contain your private copy of the library's static data. When code in the library executes on your behalf, it does so using your stack and relocates its accesses to its static data by a value stored in your stack-chunk structure by the stubs initialisation code and addressed via the stack-limit register. (This is why you must preserve the stack-limit register everywhere if you use the shared C library and call your own assembly language sub-routines.) The compiler's register allocation strategy ensures that the real dynamic cost of the relocation is almost always low: for example, by doing it once outside a loop that uses it many times.

Execution time costs

It costs only 4 cycles (0.5 μ s) per function call and a very small penalty on access to the library's static data by the library (the user program's access to the same data is unpenalised). In general, the difference in performance between using the shared C library and linking a program stand-alone with ANSILib is less than 1%. For the important Dhrystone-2.1 benchmark the performance difference cannot be measured.

How to use the C library kernel

C library structure

The C library is organised into three separate layers. At the centre is the language-independent library kernel. This is implemented in assembly language and provides basic support services, described below, to language run-time systems and, directly, to client applications.

One level out from the library kernel is a thin, C-specific layer, also implemented in assembly language. This provides compiler support functions such as structure copy, interfaces to stack-limit checking and stack extension, setjmp and longjmp support, etc. Everything above this level is written in C.

Finally, there is the C library proper. This is implemented in C and, with the exception of one module which interfaces to the library kernel and the C-specific veneer, is highly portable.

The library kernel

The library kernel provides the following facilities:

- initialisation functions
- stack management functions:
 - unwinding the stack
 - finding the current stack chunk
 - four kinds of stack extension –
 - small-frame and large-frame extension,
 - number of actual arguments known (eg Pascal), or unknown (eg C) by the callee.
- program environment functions:
 - finding the identity of the host system (RISC OS, Arthur, etc)
 - determining whether the floating point instruction set is available
 - getting the command string with which the program was invoked
 - returning the identity of the last OS error
 - reading an environment variable
 - setting an environment variable
 - invoking a sub-application
 - claiming memory to be managed by a heap manager
 - finding the name of a function containing a given address
 - finding the source language associated with code at a given address
 - determining if IRQs are enabled
 - enabling IRQs
 - disabling IRQs.
- general utility functions:
 - generic SWI interface routines
 - special SWI interfaces for certain commonly used SWIs.
- memory allocation functions:
 - allocating a block of memory in the RMA
 - extending a block of memory in the RMA
 - freeing a block of memory in the RMA.
- language support functions:
 - unsigned integer division
 - unsigned integer remainder
 - unsigned divide by 10 (much faster than general division)
 - signed integer division
 - signed integer remainder
 - signed divide by 10 (much faster than general division).

Interfacing a language run-time system to the Acorn library kernel

In order to use the kernel, a language run-time system must provide an area named `RTSK$$DATA`, with attributes `READONLY`. The contents of this area must be a `_kernel_languagedescription` as follows:

```
typedef enum { NotHandled, Handled } _kernel_HandledOrNot

typedef struct {
    int regs [16];
} _kernel_registerset;

typedef struct {
    int regs [10];
} _kernel_eventregisters;

typedef void (*PROC) (void);
typedef _kernel_HandledOrNot
    (*_kernel_trapproc) (int code, _kernel_registerset *regs);
typedef _kernel_HandledOrNot
    (*_kernel_eventproc) (int code, _kernel_registerset *regs);

typedef struct {
    int size;
    int codestart, codeend;
    char *name;
    PROC (*InitProc)(void); /* that is, InitProc returns a PROC */
    PROC FinaliseProc;
    _kernel_trapproc TrapProc;
    _kernel_trapproc UncaughtTrapProc;
    _kernel_eventproc EventProc;
    _kernel_eventproc UnhandledEventProc;
    void (*FastEventProc) (_kernel_eventregisters *);
    int (*UnwindProc) (_kernel_unwindblock *inout, char **language);
    char * (*NameProc) (int pc);
} _kernel_languagedescription;
```

Any of the procedure values may be zero, indicating that an appropriate default action is to be taken. Procedures whose addresses lie outside `[codestart...codeend]` also cause the default action to be taken.

codestart, codeend

These values describe the range of program counter (PC) values which may be taken while executing code compiled from the language. The linker ensures that this can be described with just a single base and limit pair if all code is compiled into areas with the same unique name and same attributes (conventionally, `Language$$code`, `CODE`, `READONLY`. The values required are then accessible through the symbols `Language$$code$$Base` and `Language$$code$$Limit`).

InitProc

The kernel contains the entrypoint for images containing it. After initialising itself, the kernel calls (in a random order) the `InitProc` for each language RTS present in the image. They may perform any required (language-library-specific) initialisation: their return value is a procedure to be called in order to run the main program in the image. If there is no main program in its language, an RTS should return 0. (An `InitProc` may not itself enter the main program, otherwise other language RTSs might not be initialised. In some cases, the returned procedure may be the main program itself, but mostly it will be a piece of language RTS which sets up arguments first.)

It is an error for all `InitProcs` in a module to return 0. What this means depends on the host operating system; if RISC OS, `SWI OS_GenerateError` is called (having first taken care to restore all OS handlers). If the default error handlers are in place, the difference is marginal.

FinaliseProc

On return from the entry call, or on call of the kernel's `Exit` procedure, the `FinaliseProc` of each language RTS is called (again in a random order). The kernel then removes its OS handlers and exits setting any return code which has been specified by a call of `_kernel_setreturncode`.

TrapProc, UncaughtTrapProc

On occurrence of a trap, or of a fatal error, all registers are saved in an area of store belonging to the kernel. These are the registers at the time of the instruction causing the trap, except that the PC is wound back to address that instruction rather than pointing a variable amount past it.

The PC at the time of the trap together with the call stack are used to find the `TrapHandler` procedure of an appropriate language. If one is found, it is invoked in user mode. It may return a value (`Handled` or `NotHandled`), or may not return at all. If it returns `Handled`, execution is resumed using the dumped register set (which should have been modified, otherwise resumption is likely just to repeat the trap). If it returns `NotHandled`, then that handler is marked as failed, and a search for an appropriate handler continues from the current stack frame.

If the search for a trap handler fails, then the same procedure is gone through to find a 'uncaught trap' handler.

If this too fails, it is an error. It is also an error if a further trap occurs while handling a trap. The procedure `_kernel_exित्रaphandler` is provided for use in the case the handler takes care of resumption itself (eg via `longjmp`).

(A language handler is appropriate for a PC value if $\text{LanguageCodeBase} \leq \text{PC}$ and $\text{PC} < \text{LanguageCodeLimit}$, and it is not marked as failed. Marking as ‘failed’ is local to a particular kernel trap handler invocation. The search for an appropriate handler examines the current PC, then R14, then the link field of successive stack frames. If the stack is found to be corrupt at any time, the search fails).

EventProc, UnhandledEventProc

The kernel always installs a handler for OS events and for Escape flag change. On occurrence of one, all registers are saved and an appropriate EventProc, or failing that an appropriate UnhandledEventProc is found and called. Escape pseudo-events are processed exactly like Traps. However, for ‘real’ events, the search for a handler terminates as soon as a handler is found, rather than when a willing handler is found (this is done to limit the time taken to respond to an event). If no handler is willing to claim the event, it is handed to the event handler which was in force when the program started. (The call happens in CallBack, and if it is the result of an Escape, the Escape has already been acknowledged.)

In the case of escape events, all side effects (such as termination of a keyboard read) have already happened by the time a language escape handler is called.

FastEventProc

The treatment of events by EventProc isn’t too good if what the user level handler wants to do is to buffer events (eg conceivably for the key up/down event), because there may be many events to one event handler call. The FastEventProc allows a call at the time of the event, but this is constrained to obey the rules for writing interrupt code (called in IRQ mode; must be quick; may not call SWIs or enable interrupts; must not check for stack overflow). The rules for which handler gets called in this case are rather different from those of (uncaught) trap and (unhandled) event handlers, partly because the user PC is not available, and partly because it is not necessarily quick enough. So the FastEventProc of each language in the image is called in turn (in some random order).

UnwindProc

UnwindProc unwinds one stack frame (see description of `_kernel_unwindproc` for details). If no procedure is provided, the default unwind procedure assumes that the ARM Procedure Call Standard has been used; languages should provide a procedure if some internal calls do not follow the standard.

NameProc

NameProc returns a pointer to the string naming the procedure in whose body the argument PC lies, if a name can be found; otherwise, 0.

How the run-time stack is managed and extended

The run-time stack consists of a doubly-linked list of stack chunks. The initial stack chunk is created when the run-time kernel is initialised. Currently, the size of the initial chunk is 4Kb. Subsequent requests to extend the stack are rounded up to at least this size, so the granularity of chunking of the stack is fairly coarse. However, clients may not rely on this.

Each chunk implements a portion of a descending stack. Stack frames are singly linked via their frame pointer fields within (and between) chunks. See *Appendix C: ARM procedure call standard* on page 4-397 for more details.

In general, stack chunks are allocated by the storage manager of the master language (the language in which the root procedure – that containing the language entry point – is written). Whatever procedures were last registered with `_kernel_register_allocs()` will be used (each chunk ‘remembers’ the identity of the procedure to be called to free it). Thus, in a C program, stack chunks are allocated and freed using `malloc()` and `free()`.

In effect, the stack is allocated on the heap, which grows monotonically in increasing address order.

The use of stack chunks allows multiple threading and supports languages which have co-routine constructs (such as Modula-2). These constructs can be added to C fairly easily (provided you can manufacture a stack chunk and modify the `fp`, `sp` and `sl` fields of a `jmp_buf`, you can use `setjmp` and `longjmp` to do this).

Stack chunk format

A stack chunk is described by a `_kernel_stack_chunk` data structure located at its low-address end. It has the following format:

```
typedef struct stack_chunk {
    unsigned long sc_mark;    /* == 0xf60690ff */
    struct stack_chunk *sc_next, *sc_prev;
    unsigned long sc_size;
    int (*sc_deallocate)();
} _kernel_stack_chunk;
```

`sc_mark` is a magic number; `sc_next` and `sc_prev` are forward and backward pointers respectively, in the doubly linked list of chunks; `sc_size` is the size of the chunk in bytes and includes the size of the stack chunk data structure; `sc_deallocate` is a pointer to the procedure to call to free this stack chunk – often `free()` from the C library. Note that the chunk lists are terminated by NULL pointers – the lists are not circular.

The seven words above the stack chunk structure are reserved to Acorn. The stack-limit register points 512 bytes above this (ie 560 bytes above the base of the stack chunk).

Stack extension

Support for stack extension is provided in two forms:

- fp, arguments and sp get moved to the new chunk (Pascal/Modula-2-style)
- fp is left pointing at arguments in the old chunk, and sp is moved to the new chunk (C-style).

Each form has two variants depending on whether more than 4 arguments are passed (Pascal/Modula-2-style) or on whether the required new frame is bigger than 256 bytes or not (C-style). See the appendix entitled *Appendix C: ARM procedure call standard* on page 4-397 for more details.

_kernel_stkovf_copyargs

Pascal/Modula-2-style stack extension, with some arguments on the stack (ie stack overflow in a procedure with more than four arguments). On entry, ip must contain the number of argument words on the stack.

_kernel_stkovf_copy0args

Pascal/Modula-2-style stack extension, without arguments on the stack (ie stack overflow in a procedure with four arguments or fewer).

_kernel_stkovf_split_frame

C-style stack extension, where the procedure detecting the overflow needs more than 256 bytes of stack frame. On entry, ip must contain the value of sp – the required frame size (ie the desired new sp which would be below the current stack limit).

_kernel_stkovf_split_0frame

C-style stack extension, where the procedure detecting the overflow needs 256 or fewer bytes of stack frame.

Stack chunks are deallocated on returning from procedures which caused stack extension, but with one chunk of latency. That is, one extra stack chunk is kept in hand beyond the current one, to reduce the expense of repeated call and return when the stack is near the end of a chunk; others are freed on return from the procedure which caused the extension.

Calling other programs from C

The C library procedure system() provides the means whereby a program can pass a command to the host system's command line interpreter. The semantics of this are undefined by the draft ANSI standard.

RISC OS distinguishes two kinds of commands, which we term *built-in commands* and *applications*. These have different effects. The former always return to their callers, and usually make no use of application workspace; the latter return to the previously set-up 'exit handler', and may use the currently-available application workspace. Because of these differences, `system()` exhibits three kinds of behaviour. This is explained below.

Applications in RISC OS are loaded at a fixed address specified by the application image. Normally, this is the base of application workspace, &8000. While executing, applications are free to use store between the base and end of application workspace. The end is the value returned by `SWI OS_GetEnv`. They terminate with a call of `SWI OS_Exit`, which transfers control to the current exit handler.

When a C program makes the call `system("command")` several things are done:

- The calling program and its data are copied to the top end of application workspace and all its handlers are removed.
- The current end of application workspace is set to just below the copied program and an exit handler is installed in case "command" is another application.
- "command" is invoked using `SWI OS_CLI`.

When "command" returns, either directly (if it is a built-in command) or via the exit handler (if it is an application), the caller is copied back to its original location, its handlers are re-installed and it continues, oblivious of the interruption.

The value returned by `system()` indicates

- whether the command or application was successfully invoked
- if the command is an application which obeys certain conventions, whether or not it ran successfully.

The value returned by `system` (with a non-NULL command string) is as follows:

< 0 – couldn't invoke the command or application (eg command not found);

>=0 – invoked OK and set `Sys$ReturnCode` to the returned value.

By convention, applications set the environmental variable `Sys$ReturnCode` to 0 to indicate success and to something non-0 to indicate some degree of failure. Applications written in C do this for you, using the value passed as an argument to the `exit()` function or returned from the `main()` function.

If it is necessary to replace the current application by another, use:

```
system("CHAIN:command");
```

If the first characters of the string passed to `system()` are "CHAIN:" or "chain:", the caller is not copied to the top end of application workspace, no exit handler is installed, and there can be no return (return from a built-in command is caught by the C library and turned into a `SWI OS_Exit`).

Typically, CHAIN: is used to give more memory to the called application when no return from it is required. The C compiler invokes the linker this way if a link step is required. On the other hand, the Acorn Make Utility (AMU) calls each command to be executed. Such commands include the C compiler (as both use the shared C library, the additional use of memory is minimised). Of course, a called application can call other applications using system(). A callee can even CHAIN: to another application and still, eventually, return to the caller. For example, AMU might execute:

```
system("cc hello.c");
```

to call the C compiler. In turn, cc executes:

```
system("CHAIN:link -o hello o.hello $.CLib.o.Stubs");
```

to transfer control to the linker, giving link all the memory cc had.

However, when Link terminates (calls exit(), returns from main() or aborts) it returns to AMU, which continues (providing Sys\$ReturnCode is good).

Storage management (**malloc**, **calloc**, **free**)

The aim of the storage manager is to manage the heap in as 'efficient' a manner as possible. However, 'efficient' does not mean the same to all programs and since most programs differ in their storage requirements, certain compromises have to be made.

You should always try to keep the peak amount of heap used to a minimum so that, for example, a C program may invoke another C program leaving it the maximum amount of memory. This implementation has been tuned to hold the overhead due to fragmentation to less than 50%, with a fast turnover of small blocks.

The heap can be used in many different ways. For example it may be used to hold data with a long life (persistent data structures) or as temporary work space; it may be used to hold many small blocks of data or a few large ones or even a combination of all of these allocated in a disorderly manner. The storage manager attempts to address all of these problems but like any storage manager, it cannot succeed with all storage allocation/deallocation patterns. If your program is unexpectedly running out of storage, see the section entitled *Guidelines on using memory efficiently* on page 1-348. This gives you information on the storage manager's strategy for managing the heap, and may help you to remedy the problem.

Note the following:

- The word *heap* refers to the section of memory currently under the control of the storage manager.
- All block sizes are in bytes and are rounded up to a multiple of four bytes.
- All blocks returned to the user are word-aligned.

- All blocks have an overhead of eight bytes (two words). One word is used to hold the block's length and status, the other contains a guard constant which is used to detect heap corruptions. The guard word may not be present in future releases of the ANSI C library.

Handling host errors

Calls to RISC OS can be made via one of the kernel functions, (such as `_kernel_osfind(64, "...")`). If the call causes an operating system error, the function will return the value `-2`. To find out what the error was, a call to `_kernel_last_oserror` should be made. This will return a pointer to a `_kernel_oserror` block containing the error number and any associated error string. If there has been no error since `_kernel_last_oserror` was last called, the function returns the `NULL` pointer. Some functions in the C library call `_kernel` functions, so if an C library function (such as `fopen(..., "r")`) fails, try calling `_kernel_last_oserror` to find out what the error was.

SWI Calls

SharedCLibrary_LibInitAPCS_A (SWI &80680)

This SWI interfaces an application which uses the old 'A' variant (SP=R12) of the Procedure Call Standard to the shared C library. Its use is deprecated and it should not be called in any programs. Use SharedCLibrary_LibInitAPCS_R instead.

SharedCLibrary_LibInitAPCS_R (SWI &80681)

Interfaces an application with the shared C library

On entry

R0 = pointer to list of stub descriptions each having the following format:

- +00: library chunk id (1 or 2)
- +04: entry vector base
- +08: entry vector limit
- +12: static data base
- +16: static data limit

The list is terminated by an entry with a library chunk id of -1

R1 = pointer to workspace start

R2 = pointer to workspace limit

R3 = -1

R4 = 0

R5 = -1

R6 = Bits 0 - 15 = 0

Bits 16 - 31 = Root stack size in Kilobytes

On exit

Entry vectors specified by the stubs descriptions are patched to contain branches to routines in the library.

If $R5 > R4$ on entry the users statics are copied to the bottom of the workspace specified in R1 and the Client static data offset (at byte offset +24 from the stack base) is initialised.

For each library chunk the library statics are copied either into the workspace specified in R1 if $R5 > R4$ on entry or to the static data area specified in the chunks stub description if $R5 \leq R4$.

The Library static data offset (at byte offset +20 from the stack base) is initialised.

Space for the root stack chunk is claimed from the workspace specified in R1.

R0 = value of R2 on entry

R1 = stack base

R2 = limit of space claimed from workspace passed in R1. This value should be used as the SP for the root stack chunk

R6 = library version number (currently = 5)

Interrupts

Interrupts are enabled

Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI allows you to interface an application with the shared C library without using the shared C library stubs.

LibInitAPCS_R is used by applications which use APCS_R (see *Appendix C: ARM procedure call standard* on page 4-397 for more details).

Two library chunks are currently defined.

Chunk Id 1 - The Kernel module

The Kernel module defines 48 entries, these are described in the section entitled *Library kernel functions* on page 4-275. You must reserve 48 words in your branch vector table. The words at offsets +04 and +08 of the Kernel stub description must be initialised to the start and limit (end + 1) of your vector table.

The Kernel module requires &31C bytes of static data space. You must reserve this amount of storage. The words at offsets +12 and +16 must be initialised to the start and limit (end + 1) of this storage.

Chunk Id 2 - The C library module

If you wish to use the C library module you must include the Kernel stub description before the C library stub description in the list of stubs descriptions.

The C library module defines 183 entries, these are described in the section entitled *C library functions* on page 4-288. You must reserve 183 words in your branch vector table.

The words at offsets +04 and +08 of the Kernel stub description must be initialised to the start and limit (end + 1) of your vector table.

The C library module requires &B48 bytes of static data space. You must reserve this amount of storage. The words at offsets +12 and +16 must be initialised to the start and limit (end + 1) of this storage. This storage must be contiguous with that for the Kernel module.

Calling library functions

Before calling any library functions you must call the kernel function `_kernel_init` (entry no. 0). For details on how to call these functions refer to their entries in the section entitled *Library kernel functions* on page 4-275.

SP, SL and FP must be set up before calling any library function. `_kernel_init` initialises these for the root stack chunk passed to it.

If you wish to call C library functions you must pass a suitable kernel language description block to `_kernel_init`. For details on the format of a kernel language description block refer to the section entitled *Interfacing a language run-time system to the Acorn library kernel* on page 4-242.

To call C library functions the fields of the kernel language description block must be as follows:

size	The size of this structure in bytes (24 - 52 depending on the number of entries in this block).
codestart, codelimit	These two words should be set to the start and limit of an area which is to be treated as C code with respect to trap and event handling. Both these values may be set to 0 in which case no traps or events will be passed to the trap or event handler described in this language description block.
name	This must contain a pointer to the 0 terminated string "C".
InitProc	Pointer to your initialisation procedure. Your initialisation procedure must call <code>_clib_initialise</code> (entry no. 20). For details on how to call <code>_clib_initialise</code> refer to its entry in the section entitled <i>C library functions</i> on page 4-288. It should then load R0 with the address at which execution is to continue at the end of initialisation.
FinaliseProc	Pointer to your finalisation procedure. This may contain 0.

The remainder of the entries are optional and may omitted. You must set the size field correctly if omitting entries. If all optional entries are omitted the size field should be set to 24.

Related SWIs

SharedCLibrary_LibInitAPCS_A (SWI &80680)

Related vectors

None

SharedCLibrary_LibInitModule (SWI &80682)

Interfaces a module with the shared C library

On entry

R0 = pointer to list of stub descriptions each having the following format:

- +00: library chunk id (1 or 2)
- +04: entry vector base
- +08: entry vector limit
- +12: static data base
- +16: static data limit

The list is terminated by an entry with a library chunk id of -1

R1 = pointer to workspace start

R2 = pointer to workspace limit

R3 = base of area to be zero-initialised

R4 = pointer to start of static data

R5 = pointer to limit of static data

R6 = Bits 0 - 15 = 0

Bits 16 - 31 = Root stack size in Kilobytes

On exit

Entry vectors specified by the stubs descriptions are patched to contain branches to routines in the library.

If $R5 > R4$ on entry the users statics are copied to the bottom of the workspace specified in R1 and the Client static data offset (at byte offset +24 from the stack base) is initialised.

For each library chunk the library statics are copied either into the workspace specified in R1 if $R5 > R4$ on entry or to the static data area specified in the chunks stub description if $R5 \leq R4$.

The Library static data offset (at byte offset +20 from the stack base) is initialised.

Space for the root stack chunk is claimed from the SVC stack.

R0 = value of R2 on entry

R1 = stack base

R2 = limit of space claimed from workspace passed in R1

R6 = library version number (currently = 5)

Note: You must save the words at offsets +20 and +24 from the returned stack base. You must do this before exiting your module initialisation code. These words contain the shared libraries static data offset and the client static data offset (the offset you must use when accessing your static data). These must be restored in the static data offset locations at offsets +00 and +04 from the base of the SVC stack when you are re-entering the module in SVC mode (e.g. in a SWI handler). When restoring the static data offsets you must save the previous static data offsets around the module entry.

Interrupts

Interrupts are enabled
Fast interrupts are enabled

Processor mode

Processor is in SVC mode

Re-entrancy

SWI is re-entrant

Use

This SWI allows you to interface a module with the shared C library without using the shared C library stubs.

SharedCLibrary_LibInitModule is used by modules, which must use APCS_R, and must be called in the module Initialisation code.

Two library chunks are currently defined.

Chunk Id 1 - The Kernel module

The Kernel module defines 48 entries, these are described in the section entitled *Library kernel functions* on page 4-275. You must reserve 48 words in your branch vector table. The words at offsets +04 and +08 of the Kernel stub description must be initialised to the start and limit (end + 1) of your vector table.

The Kernel module requires &31C bytes of static data space. You must reserve this amount of storage. The words at offsets +12 and +16 must be initialised to the start and limit (end + 1) of this storage.

Chunk Id 2 - The C library module

If you wish to use the C library module you must include the Kernel stub description before the C library stub description in the list of stubs descriptions.

The C library module defines 183 entries, these are described in the section entitled *C library functions* on page 4-288. You must reserve 183 words in your branch vector table.

The words at offsets +04 and +08 of the Kernel stub description must be initialised to the start and limit (end + 1) of your vector table.

The C library module requires &B48 bytes of static data space. You must reserve this amount of storage. The words at offsets +12 and +16 must be initialised to the start and limit (end + 1) of this storage. This storage must be contiguous with that for the Kernel module.

Calling library functions

Before calling any library functions you must call the kernel function `_kernel_moduleinit` (entry no. 38). For details on how to call these functions refer to their entries in the section entitled *Library kernel functions* on page 4-275.

SP, SL and FP must be set up before calling any library function. `_kernel_init` initialises these for the root stack chunk passed to it.

If you wish to call C library functions you must pass a suitable kernel language description block to `_kernel_init`. For details on the format of a kernel language description block refer to the section entitled *Interfacing a language run-time system to the Acorn library kernel* on page 4-242.

To call C library functions the fields of the kernel language description block must be as follows:

size	The size of this structure in bytes (24 - 52 depending on the number of entries in this block).
codestart, codelimit	These two words should be set to the start and limit of an area which is to be treated as C code with respect to trap and event handling. Both these values may be set to 0 in which case no traps or events will be passed to the trap or event handler described in this language description block.
name	This must contain a pointer to the 0 terminated string "C".
InitProc	Pointer to your initialisation procedure. Your initialisation procedure must call <code>_clib_initialise</code> (entry no. 20). For details on how to call <code>_clib_initialise</code> refer to its entry in the section entitled <i>C library functions</i> on page 4-288. It should then load R0 with the address at which execution is to continue at the end of initialisation.
FinaliseProc	Pointer to your finalisation procedure. This may contain 0.

The remainder of the entries are optional and may omitted. You must set the size field correctly if omitting entries. If all optional entries are omitted the size field should be set to 24.

Accessing shared library data

The following items of data are exported from the shared library data and may be used in your programs.

Name	Offset	Notes
errno	&0000	The variable errno is set whenever certain error conditions arise in the C library. These error conditions are described in the section 'errno' on page 4-307.
stdin	&0004	These three variables contain the standard C library FILE structures stdin, stdout and stderr. The address of these variables may be passed to any C library function which accept a FILE * argument. For an example of their use see the call to 'fputs' in the module example.
stdout	&002C	
stderr	&0054	
ctype	&0290	This is a 256 byte array containing an 8 bit mask for each character in the range 0 to 255. Each bit defines some aspect of the character as follows: bit 0 character is a whitespace character bit 1 character is a punctuation character bit 2 character is a blank (' ') bit 3 character is a lowercase letter bit 4 character is an uppercase letter bit 5 character is a decimal digit bit 6 character is a control character bit 7 character is one of the characters A, B, C, D, E, F or a, b, c, d, e, f This table is initialised for the C locale; it may be changed by calls to the 'setlocale' function.

Note: The offsets given above are offsets into the C library statics. These must be preceded immediately by the kernel statics, which are 800 (&31C) bytes long. To convert offsets in the C library statics to offsets in the library statics add 800 (&31C).

If you are accessing static data within a program (i.e. code which uses SharedCLibrary_LibInitAPCS_R) you can access the static data directly in your own static data area definition. If, however, you are accessing static data from within a module (using SharedCLibrary_LibInitModule) you must use the add the client static data relocation to the address in your own static data area definition to obtain the

true address of the static data. If you wish your module to be multiply instantiable or rommable you must add this relocation when accessing your own static data, not just when accessing the libraries static data.

The client static data relocation is stored at offset -536 (-&218) from the SL register (R10).

For an example of how to use the static data relocation see the call to ‘fputs’ in the module example.

Related SWIs

None

Related vectors

None

Example programs

Calling the shared C library

```

; This example demonstrates how to call the shared C library.
; It is written for the ObjAsm assembler supplied with the Software
; Developers Toolkit (SDT) and the Desktop Development Environment (DDE).
;
r0      RN   0
r1      RN   1
r2      RN   2
r3      RN   3
r4      RN   4
r5      RN   5
r6      RN   6
sp      RN  13
lr      RN  14
pc      RN  15

_l_kernel_initl    EQU  0 * 4      ; Offsets in kernel vector table

_l_clib_initialisel EQU  20 * 4    ; Offsets in C vector table
fopen             EQU  87 * 4
fprintf           EQU  92 * 4
fclose            EQU  85 * 4

OS_GenerateError EQU  &2b
OS_Exit           EQU  &11

SharedCLibrary_LibInitAPCS_R EQU &80681

IMPORT !Image$$RO$$Base ; Linker defined symbol giving
; start of image.
AREA  printf, CODE, READONLY

ENTRY

ADR   r0, stubs
ADRL r1, workspace
ADD  r2, r1, #32 * 1024 ; 32K workspace. A real program
MOV  r3, #-1           ; would use OS_ChangeEnvironment
MOV  r4, #0            ; to find the memorylimit.
MOV  r5, #-1
MOV  r6, #&00080000
SWI  SharedCLibrary_LibInitAPCS_R
MOV  r4, r0
ADR  r0, kernel_init_block
MOV  r3, #0
B   kernel_vectors + _l_kernel_initl ; Continues at c_init below

```

```

stubs
    DCD 1
    DCD kernel_vectors
    DCD kernel_vectors_end
    DCD kernel_statics
    DCD kernel_statics_end DCD 2
    DCD clib_vectors
    DCD clib_vectors_end
    DCD clib_statics
    DCD clib_statics_end

    DCD -1

kernel_init_block
    DCD !Image$$RO$$Base!
    DCD rts_block
    DCD rts_block_end

rts_block
    DCD rts_block_end - rts_block
    DCD 0
    DCD 0
    DCD c_str
    DCD c_init
    DCD 0

rts_block_end

c_str    DCB  "C", 0          ; Must be "C" for CLib to finalise
        ALIGN                ; properly.

c_init   MOV  r0, sp
        MOV  r1, #0
        MOV  r2, #0
        STMDB sp!, {r}
        BL   clib_vectors + !clib_initialise!
        ADR  r0, c_run        ; Continue at c_run below
        LDMIA sp!, {pc}^

c_run    ADR  r0, outfile
        ADR  r1, access
        BL   clib_vectors + fopen
        CMP  r0, #0
        ADREQ r0, Err_Open    ; Will actually say
        SWIEQ OS_GenerateError ; Uncaught trap: Error opening ...
        MOV  r4, r0
        ADR  r1, format
        BL   clib_vectors + fprintf
        MOV  r0, r4
        BL   clib_vectors + fclose
        CMP  r0, #0
        ADRNE r0, Err_Close
        SWINE OS_GenerateError ; Uncaught trap: Error writing ...
        SWI  OS_Exit

```

```
outfile    DCB    "OutFile", 0
access    DCB    "w", 0
format    DCB    "Sample string printed from asm using fprintf!", 10, 0
          ALIGN

Err_Open   DCD    &1000
          DCB    "Error opening OutFile", 0
          ALIGN

Err_Close  DCD    &1001
          DCB    "Error writing OutFile", 0
          ALIGN

kernel_vectors % 48 * 4
kernel_vectors_end

clib_vectors % 183 * 4
clib_vectors_end

kernel_statics % &31c
kernel_statics_end

clib_statics % &b48
clib_statics_end

workspace          ; Start of workspace at end of app.

          END
```

Calling the shared C library from a module

```
; This example demonstrates how to call the shared C library from a module.
; It is written for the ObjAsm assembler supplied with the Software
; Developers Toolkit (SDT) and the Desktop Development Environment (DDE)
r0    RN    0
r1    RN    1
r2    RN    2
r3    RN    3
r4    RN    4
r5    RN    5
r6    RN    6
r7    RN    7
r8    RN    8
r9    RN    9
r10   RN    10
r11   RN    11
r12   RN    12

sl    RN    10
fp    RN    11
sp    RN    13
lr    RN    14
pc    RN    15

swibase    EQU    &88000
V_Bit     EQU    1:SHL:28
```

```

Module_Claim EQU 6

Service_Error EQU &06
Service_Help EQU &09

XOS_Module EQU &2001e
XSharedCLibrary_LibInitModule EQU &80682

OS_WriteS EQU 1
OS_Exit EQU &11

    ^ 0 ; Offsets in module workspace
size # 4 ; Size of this block
libreloc # 4 ; Offset for accessing library's statics
clientreloc # 4 ; Offset for accessing our statics
ws_size # 0

Lib_Offset EQU 20 ; Offset of library relocation offset
; from base of stack.
SL_Lib_Offset EQU 540 ; Negative offset of library relocation
; offset from SL register
Client_Offset EQU 24 ; Offset of client relocation offset
SL_Client_Offset EQU 536 ; Negative offset of client relocation
; offset from SL register

|_kernel_command_string| EQU 7 * 4
|_kernel_moduleinit| EQU 38 * 4
|_kernel_entermodule| EQU 42 * 4

|_main| EQU 18 * 4
|_clib_initialise| EQU 20 * 4
atexit EQU 71 * 4
printf EQU 91 * 4
fputs EQU 104 * 4
putchar EQU 111 * 4
|_clib_finalisemodule| EQU 179 * 4

IMPORT |__RelocCode| ; Linker supplied relocation routine
IMPORT |Image$$RO$$Base| ; Linker defined base / limit symbols
IMPORT |Image$$RW$$Base|
IMPORT |Image$$RW$$Limit|
IMPORT |Image$$ZI$$Base|

AREA module_code, CODE, READONLY

module_base
DCD start - module_base
DCD init - module_base
DCD terminate - module_base
DCD service - module_base
DCD title - module_base
DCD help - module_base
DCD cmdtbl - module_base
DCD swibase
DCD swicode - module_base
DCD switbl - module_base

```

```

title DCB "SLClient", 0
help DCB "SLClient", 9, "1.00 (11-Dec-91)", 0
      ALIGN

base DCD IImage$$RW$$BaseI
limit DCD IImage$$RW$$LimitI
zi_base DCD IImage$$ZI$$BaseI

cmdtbl DCB "SLClient_Command", 0
      ALIGN
      DCD cmdcode - module_base
      DCB 0
      DCB &ff
      DCB 255
      DCB 0
      DCD 0 ; No syntax message
      DCD 0 ; No help message

switbl DCB "SLClient", 0
      DCB "SWI", 0 ; SLClient_SWI
      DCB 0
      ALIGN

init STMDB sp!, {r7-r11, lr} ; Save only regs that need saving
MOV sl, sp, LSR #20 ; Get base of SVC stack in sl.
MOV sl, sl, LSL #20
LDMIA sl, {r4, r5} ; Save old relocation modifiers
STMDB sp!, {r4, r5} ; from base of SVC stack
BL __RelocCodeI ; Relocate module
MOV r0, #Module_Claim
LDR r4, base
LDR r5, limit
SUB r3, r5, r4
ADD r3, r3, #ws_size
SWI XOS_Module
MOV r9, r12
STR r2, [r12] ; Set private word
MOV r12, r2
STR r3, [r12] ; First word of block is size of block
ADR r0, stubs
ADD r1, r12, #ws_size
ADD r2, r12, r3
LDR r3, zi_base
MOV r6, #4 ;SHL: 16
SWI XSharedCLibrary_LibInitModule
ADD r8, r1, #Lib_Offset
LDMIA r8, {r7, r8} ; Get Lib and Client reloc. offset
STMIB r12, {r7, r8} ; Save in work area
ADR r0, kernel_init_block
BL call_moduleinit
STMDB sp!, {r9} ; Save workspace pointer
BL clib_vectors + l_clib_initialiseI
LDMIA sp!, {r2}

```

```

ADD    r0, sp, #(10-7+2)*4 ; Point to R10 on stack
LDMIA  r0, {r0, r1}
BL     user_init
MOV    sl, sp, LSR #20 ; Get base of SVC stack in sl.
MOV    sl, sl, LSL #20
LDMIA  sp!, {r4, r5}
STMIA  sl, {r4, r5}
LDMIA  sp!, {r7-r11, lr}
CMPS   r0, #0
BICEQS pc, lr, #V_Bit
ORRS   pc, lr, #V_Bit

; _kernel_moduleinit expects the return address to be in the first word on the
; stack rather than in LR. This function sets up the return address correctly.
call_moduleinit
    STMDB sp!, {lr}
    B     kernel_vectors + |_kernel_moduleinit|

terminate
    STMDB sp!, {r7-r11, lr} ; Save only regs that need saving
    MOV   sl, sp, LSR #20 ; Get base of SVC stack in sl.
    MOV   sl, sl, LSL #20
    LDMIA sl, {r4, r5} ; Save old relocation modifiers
    MOV   r0, r12 ; Set up private word pointer for
    ; _clib_finalisemodule
    LDR   r12, [r12] ; Pointer to static data
    LDMIB r12, {r11, r12}
    STMIA sl, {r11, r12} ; Set up relocation modifiers
    ADD   sl, sl, #SL_Lib_Offset
    MOV   fp, #0 ; FP = 0 => end of linked stack frames
    ; so backtrace stops here
    BL   clib_vectors + |_clib_finalisemodule|
    MOV   sl, sp, LSR #20
    MOV   sl, sl, LSL #20
    STMIA sl, {r4, r5} ; Restore old relocation modifiers
    LDMIA sp!, {r7-r11, pc}^

start  ADR   r0, kernel_init_block
    MOV   r8, r12
    MOV   r12, #-1
    MOV   r6, #4 * 1024
    B     kernel_vectors + |_kernel_entermodule|

c_init STMDB sp!, {lr}
    BL   clib_vectors + |_clib_initialise|
    ADR   r0, c_run ; Continue at c_run below
    LDMIA sp!, {pc}

c_run  BL   kernel_vectors + |_kernel_command_string|
    ADR   r1, user_run ; Continue at user_run below
    B     clib_vectors + |_main|

```

```

cmdcode STMDB sp!, {r10, r11, lr}
MOV sl, sp, LSR #20 ; Get base of SVC stack
MOV sl, sl, ASL #20
LDMIA sl, {r4, r5} ; Save old relocation modifiers in R4, R5
LDR r12, [r12]
LDMIB r12, {r11, r12} ; Set up our relocation modifiers
STMIA sl, {r11, r12}
ADD sl, sl, #SL_Lib_Offset ; Set up stack limit for SVC stack
MOV fp, #0 ; Stop backtrace here
BL user_cmd ; Call APCS user_cmd routine
MOV sl, sp, LSR #20
MOV sl, sl, ASL #20 ; Get base of SVC stack again
STMIA sl, {r4, r5} ; Restore old relocation modifiers
LDMIA sp!, {r10, r11, lr}
CMP r0, #0 ; Set V bit on R0 and return
BICEQS pc, lr, #V_Bit
ORRS pc, lr, #V_Bit

swicode STMDB sp!, {r0-r9, lr} ; Set up regset on SVC stack
MOV sl, sp, LSR #20 ; Get base of SVC stack
MOV sl, sl, ASL #20
LDMIA sl, {r8, r9} ; Save old relocation modifiers in R8, R9
MOV r0, r11
MOV r1, sp ; Pointer to regs on stack
MOV r2, r12
LDR r12, [r12]
LDMIB r12, {r11, r12} ; Set up relocation modifiers
STMIA sl, {r11, r12}
ADD sl, sl, #SL_Lib_Offset ; Set up stack limit for SVC stack
MOV fp, #0 ; Stop backtrace here
BL user_swi ; Call APCS user_swi routine
MOV sl, sp, LSR #20 ; Get base of SVC stack again
MOV sl, sl, ASL #20
STMIA sl, {r8, r9} ; Restore old relocation modifiers
CMP r0, #0 ; Set R0 on stack to error pointer
STRNE r0, [sp] ; if error on return.
LDMIA sp!, {r0-r9, lr}
BICEQS pc, lr, #V_Bit ; Set V bit on R0 and return.
ORRS pc, lr, #V_Bit

service TEQ r1, #Service_Help ; Check service nos. first for speed
TEQNE r1, #Service_Error
MOVNES pc, lr
STMDB sp!, {r0-r9, sl, fp, lr} ; Set up regset on SVC/IRQ stack
MOV r0, r1
MOV r1, sp ; Pointer to regs on stack
MOV r6, pc ; Save old mode
BIC lr, r6, #3 ; To SVC mode from SVC/IRQ mode
TEQP lr, #3
MOV r0, r0 ; NOP after mode change
MOV fp, #0 ; Stop backtrace
MOV r7, lr ; Save SVC lr if entered in IRQ mode
MOV sl, sp, LSR #20 ; Get base of SVC stack
MOV sl, sl, ASL #20
LDMIA sl, {r8, r9} ; Save old relocation modifiers in R8, R9

```

```

MOV    r2, r12
LDR    r12, [r12]
LDMIB  r12, {r11, r12} ; Set up relocation modifiers
STMIA  sl, {r11, r12}
ADD    sl, sl, #SL_Lib_Offset ; Set up stack limit for SVC stack
BL     user_service ; Call APCS user_service routine
MOV    lr, r7 ; Restore SVC lr
TEQP   r6, #0 ; Back to entry mode
MOV    r0, r0 ; NOP after mode change
MOV    sl, sp, LSR #20 ; Get base of SVC stack
MOV    sl, sl, ASL #20
STMIA  sl, {r8, r9} ; Restore old relocation modifiers
LDMIA  sp!, {r0-r9, sl, fp, pc}^

; _kernel_oserror *user_init(char *cmd_tail, int base, void *pw);
user_init
    STMDB sp!, {r4, r9, lr}
    LDR  r9, [sl, #-SL_Client_Offset] ; Get Client relocation
    MOV  r4, r0
    ADR  r0, format
    ADR  r1, init_str
    BL  clib_vectors + printf
    ADR  r0, cmd_format
    LDR  r1, stdout ; Address stdout in library statics
    ADD  r1, r1, r9 ; Add client relocation
    BL  clib_vectors + fputs
10    LDRB r0, [r4], #1
    CMP  r0, #32
    MOVCC r0, #10
    BL  clib_vectors + putchar
    BCS %B10
    ADR  r0, user_exit ; Set up atexit handler
    BL  clib_vectors + atexit
    MOV  r0, #0
    LDMIA sp!, {r4, r9, pc}^

stdout DCD  clib_statics + &2c

; void user_exit(void);
user_exit
    STMDB sp!, {lr}
    ADR  r0, format
    ADR  r1, exit_str
    BL  clib_vectors + printf
    LDMIA sp!, {pc}^

```

```
; int user_run(int argc, char **argv);
user_run
    STMDB sp!, {r4, r5, r6, lr}
    MOV   r4, r0
    MOV   r5, r1
    ADR   r0, format
    ADR   r1, run_str
    BL   clib_vectors + printf
    ADR   r0, argc_format
    MOV   r1, r4
    BL   clib_vectors + printf
    MOV   r6, #0
10    CMP   r6, r4
    ADRCC r0, argv_format
    MOVCC r1, r6
    LDRCC r2, [r5, r6, LSL #2]
    BLCC  clib_vectors + printf
    ADDCC r6, r6, #1
    BCC   %B10
    MOV   r0, #0
    LDMIA sp!, {r4, r5, r6, pc}^

; _kernel_oseerror *user_cmd(char *args, int argc);
user_cmd
    STMDB sp!, {r4, r5, lr}
    MOV   r4, r0
    MOV   r5, r1
    ADR   r0, format
    ADR   r1, cmd_str
    BL   clib_vectors + printf
    ADR   r0, args_format
    MOV   r1, r5
    BL   clib_vectors + printf
10    LDRB  r0, [r4], #1
    CMP   r0, #32
    MOVCC r0, #10
    BL   clib_vectors + putchar
    BCS   %B10
    MOV   r0, #0
    LDMIA sp!, {r4, r5, pc}^

; _kernel_oseerror *user_swi(int swi_no, _kernel_swi_regs *r, void *pw);
user_swi
    STMDB sp!, {lr}
    ADR   r0, format
    ADR   r1, swi_str
    BL   clib_vectors + printf
    MOV   r0, #0
    LDMIA sp!, {pc}^
```

```

; void user_service(int service_no, _kernel_swi_regs *r, void *pw);
user_service
    STMDB    sp!, {lr}
    CMP     r0, #Service_Help
    ADR     r0, format
    ADREQ   r1, help_str
    ADRNE   r1, error_str
    BL     clib_vectors + printf
    LDMIA   sp!, {pc}^

format      DCB    "In %s code", 10, 0
            ALIGN
argc_format DCB    "argc = %d", 10, 0
            ALIGN
argv_format DCB    "argv[%d] = %s", 10, 0
            ALIGN
args_format DCB    "argc = %d, args = ", 0
            ALIGN
cmd_format  DCB    "Command tail = ", 0
            ALIGN
init_str    DCB    "initialisation", 0
            ALIGN
exit_str    DCB    "exit", 0
            ALIGN
run_str     DCB    "run", 0
            ALIGN
cmd_str     DCB    "command", 0
            ALIGN
swi_str     DCB    "swi", 0
            ALIGN
help_str    DCB    "help", 0
            ALIGN
error_str   DCB    "error", 0
            ALIGN

stubs
    DCD    1
    DCD    kernel_vectors
    DCD    kernel_vectors_end
    DCD    kernel_statics
    DCD    kernel_statics_end

    DCD    2
    DCD    clib_vectors
    DCD    clib_vectors_end
    DCD    clib_statics
    DCD    clib_statics_end

    DCD    -1

kernel_init_block
    DCD    !Image$$RO$$Basel
    DCD    rts_block
    DCD    rts_block_end

```

```
    rts_block
        DCD    rts_block_end - rts_block
        DCD    0
        DCD    0
        DCD    c_str
        DCD    c_init
        DCD    0
    rts_block_end

    c_str DCB    "C", 0
        ALIGN

    kernel_vectors %    48 * 4
    kernel_vectors_end

    clib_vectors %    183 * 4
    clib_vectors_end

; Unlike the application example the kernel statics and clib statics must be in
; a data area otherwise the data size calculation above (using Image$$RW$$Base
; & Image$$RW$$Limit does not work.
;
; Ideally this would be a zero init area of appropriate size but the assembler
; doesn't support zero init areas.
        AREA    module_data

    kernel_statics %    &31c
    kernel_statics_end

    clib_statics %    &b48
    clib_statics_end

        END
```

Library kernel functions

The library kernel functions are grouped under the following headings:

- initialisation functions
- stack management functions
- program environment functions
- general utility functions
- memory allocation functions
- language support functions.

Index of library kernel functions by entry number

entry no.	Name	on page
0	<code>_kernel_init</code>	page 4-278
1	<code>_kernel_exit</code>	page 4-281
2	<code>_kernel_setreturncode</code>	page 4-281
3	<code>_kernel_exittraphandler</code>	page 4-282
4	<code>_kernel_unwind</code>	page 4-281
5	<code>_kernel_procname</code>	page 4-281
6	<code>_kernel_language</code>	page 4-281
7	<code>_kernel_command_string</code>	page 4-281
8	<code>_kernel_hostos</code>	page 4-282
9	<code>_kernel_swi</code>	page 4-283
10	<code>_kernel_osbyte</code>	page 4-284
11	<code>_kernel_osrdch</code>	page 4-284
12	<code>_kernel_oswrch</code>	page 4-284
13	<code>_kernel_osbget</code>	page 4-284
14	<code>_kernel_osbput</code>	page 4-284
15	<code>_kernel_osgbpb</code>	page 4-284
16	<code>_kernel_osword</code>	page 4-284
17	<code>_kernel_osfind</code>	page 4-285
18	<code>_kernel_osfile</code>	page 4-285
19	<code>_kernel_osargs</code>	page 4-285
20	<code>_kernel_oscli</code>	page 4-285
21	<code>_kernel_last_oserror</code>	page 4-282
22	<code>_kernel_system</code>	page 4-285
23	<code>_kernel_getenv</code>	page 4-282
24	<code>_kernel_setenv</code>	page 4-282
25	<code>_kernel_register_allocs</code>	page 4-286
26	<code>_kernel_alloc</code>	page 4-286
27	<code>_kernel_stkovf_split_0frame</code>	page 4-280

entry no.	Name	on page
28	<code>_kernel_stkovf_split</code>	page 4-280
29	<code>_kernel_stkovf_copyargs</code>	page 4-280
30	<code>_kernel_stkovf_copy0args</code>	page 4-280
31	<code>_kernel_udiv</code>	page 4-286
32	<code>_kernel_urem</code>	page 4-287
33	<code>_kernel_udiv10</code>	page 4-287
34	<code>_kernel_sdiv</code>	page 4-287
35	<code>_kernel_srem</code>	page 4-287
36	<code>_kernel_sdiv10</code>	page 4-287
37	<code>_kernel_fpavailable</code>	page 4-282
38	<code>_kernel_moduleinit</code>	page 4-279
39	<code>_kernel_irqs_on</code>	page 4-283
40	<code>_kernel_irqs_off</code>	page 4-283
41	<code>_kernel_irqs_disabled</code>	page 4-283
42	<code>_kernel_entermodule</code>	page 4-279
43	<code>_kernel_escape_seen</code>	page 4-282
44	<code>_kernel_current_stack_chunk</code>	page 4-280
45	<code>_kernel_swi_c</code>	page 4-283
46	<code>_kernel_register_slotextend</code>	page 4-286
47	<code>_kernel_raise_error</code>	page 4-282

Index of library kernel functions by function name

Name	entry no.	on page
<code>_kernel_alloc</code>	26	page 4-286
<code>_kernel_command_string</code>	7	page 4-281
<code>_kernel_current_stack_chunk</code>	44	page 4-280
<code>_kernel_entermodule</code>	42	page 4-279
<code>_kernel_escape_seen</code>	43	page 4-282
<code>_kernel_exit</code>	1	page 4-281
<code>_kernel_exittraphandler</code>	3	page 4-282
<code>_kernel_fpavailable</code>	37	page 4-282
<code>_kernel_getenv</code>	23	page 4-282
<code>_kernel_hostos</code>	8	page 4-282
<code>_kernel_init</code>	0	page 4-278
<code>_kernel_irqs_disabled</code>	41	page 4-283
<code>_kernel_irqs_off</code>	40	page 4-283
<code>_kernel_irqs_on</code>	39	page 4-283
<code>_kernel_language</code>	6	page 4-281
<code>_kernel_last_oserror</code>	21	page 4-282
<code>_kernel_moduleinit</code>	38	page 4-279
<code>_kernel_osargs</code>	19	page 4-285

Name	entry no.	on page
<code>_kernel_osbget</code>	13	page 4-284
<code>_kernel_osbput</code>	14	page 4-284
<code>_kernel_osbyte</code>	10	page 4-284
<code>_kernel_oscli</code>	20	page 4-285
<code>_kernel_osfile</code>	18	page 4-285
<code>_kernel_osfind</code>	17	page 4-285
<code>_kernel_osgbpb</code>	15	page 4-284
<code>_kernel_osrdch</code>	11	page 4-284
<code>_kernel_osword</code>	16	page 4-284
<code>_kernel_oswrch</code>	12	page 4-284
<code>_kernel_procname</code>	5	page 4-281
<code>_kernel_raise_error</code>	47	page 4-282
<code>_kernel_register_allocs</code>	25	page 4-286
<code>_kernel_register_slotextend</code>	46	page 4-286
<code>_kernel_sdiv</code>	34	page 4-287
<code>_kernel_sdiv10</code>	36	page 4-287
<code>_kernel_setenv</code>	24	page 4-282
<code>_kernel_setreturncode</code>	2	page 4-281
<code>_kernel_srem</code>	35	page 4-287
<code>_kernel_stkovf_copy0args</code>	30	page 4-280
<code>_kernel_stkovf_copyargs</code>	29	page 4-280
<code>_kernel_stkovf_split</code>	28	page 4-280
<code>_kernel_stkovf_split_0frame</code>	27	page 4-280
<code>_kernel_swi</code>	9	page 4-283
<code>_kernel_swi_c</code>	45	page 4-283
<code>_kernel_system</code>	22	page 4-285
<code>_kernel_udiv</code>	31	page 4-286
<code>_kernel_udiv10</code>	33	page 4-287
<code>_kernel_unwind</code>	4	page 4-281
<code>_kernel_urem</code>	32	page 4-287

The following structure is common to all library kernel functions:

```
typedef struct {
    int ernum; /* error number */
    char errmsg[252]; /* error message (zero terminated) */
} _kernel_oseerror;
```

Initialisation functions

Entry no. 0: `_kernel_init`

On entry

R0 = Pointer to kernel init block having the following format
+00: Image base (e.g. the value of the linker symbol `Image$$RO$$Base`)
+04: pointer to start of language description blocks
+08: pointer to end of language description blocks
R1 = base of root stack chunk (value returned in R1 from `LibInitAPCS_A` or `LibInitAPCS_R`)
R2 = top of root stack chunk (value returned in R2 from `LibInitAPCS_A` or `LibInitAPCS_R`)
R3 = 0 for application
1 for module
R4 = end of workspace

On exit

Does not return. Control is regained through the procedure pointer returned in R0 by one of the language initialisation procedures (i.e. control is passed to the run code of the language).

This call does not obey the APCS. All registers are altered. The APCS_R SL, FP and SP (R10, R11 and R13) are set up. LR does not contain a valid return address when control is passed to the run entry.

This function must be called by any client which calls `LibInitAPCS_A` or `LibInitAPCS_R`. Modules should call this entry in their run entry.

The words at offsets +04 and +08 from R0 describe an area containing at least one language description block. Any number of language description blocks may be present. The size field of each block must be the offset to the next language description block.

The command line is copied to an internal buffer at the top of the root stack chunk. To set a command line call `SWI OS_WriteEnv`. RISC OS sets up a command line before running your application or entering your module.

Exit, Error, CallBack, Escape, Event, UpCall, Illegal Instruction, Prefetch Abort, Data Abort and Address Exception handlers are set up.

Initial default alloc and free procs for use during stack extension are set up. These should be replaced with your own alloc and free procs as soon as possible.

The kernel's workspace pointers are initialised to the values contained in R1 and R4. Note that it is assumed the root stack chunk resides at the base of the workspace area.

A small stack (159 words) for use during stack extension is claimed from the workspace following R2 (i.e. 159 words are claimed from R2 upwards).

Note: `_kernel_init` does not check that there is sufficient space in the workspace to claim this area. You must ensure there is sufficient space before calling `_kernel_init`.

The availability of floating point is determined (by calling `SWI FPE_Version`).

If executing under the desktop the initial wimplot size is determined by reading the Application Space handler.

The initialisation for each language is called, then the run code if any is called. If no run code is present the error No main program is generated.

Entry no. 38: `_kernel_moduleinit`

On entry

R0 = pointer to kernel init block as described in `_kernel_init` on page 4-278

R1 = pointer to base of SVC stack (as returned by `SWI LibInitModule`)

On exit

This call does not obey the APCS.

It assumes that LR has already been pushed on the stack, and so returns to the address on top of the stack (ie the address pointed to by SP), rather than to the address contained in LR on entry. The stack pointer is incremented by 4. See the section entitled *Calling the shared C library from a module* on page 4-266 for an example.

On exit SL points to R1 on entry + 560.

R0, R1, R2 and R12 are indeterminate.

The kernel init block is copied for later use. The Image base is ignored.

The functions `_kernel_RMAalloc` and `_kernel_RMAfree` are established as the default alloc and free procs for use during stack extension.

You should call this function after calling `SWI LibInitModule`.

Entry no. 42: `_kernel_entermodule`

On entry

R0 = pointer to kernel init block as described in `_kernel_init` on page 4-278

R6 = requested root stack size

R8 = modules private word pointer

R12 = -1

On exit

Does not return.

Control is regained through the procedure pointer returned in R0 by one of the language initialisation procedures.

The private word must point to the module workspace word which must contain the application base, the shared library static offset, and the client static offset in words 0, 1 and 2 (the application base is ignored for modules).

After claiming workspace from the application space and claiming a root stack from this `_kernel_entermodule` calls `_kernel_init`.

Stack management functions

Entry no. 27: `_kernel_stkovf_split_0frame`

This function is described in the section entitled *How the run-time stack is managed and extended* on page 4-243.

Entry no. 28: `_kernel_stkovf_split`

This function is described in the section entitled *How the run-time stack is managed and extended* on page 4-243.

Entry no. 29: `_kernel_stkovf_copyargs`

This function is described in the section entitled *How the run-time stack is managed and extended* on page 4-243.

Entry no. 30: `_kernel_stkovf_copy0args`

This function is described in the section entitled *How the run-time stack is managed and extended* on page 4-243.

```
typedef struct stack_chunk {
    unsigned long sc_mark; /* == 0xf60690ff */
    struct stack_chunk *sc_next, *sc_prev;
    unsigned long sc_size;
    int (*sc_deallocate)()
} _kernel_stack_chunk;
```

Entry no. 44: `_kernel_stack_chunk *_kernel_current_stack_chunk(void)`

Returns a pointer to the current stack chunk.

```
typedef struct {
    int r4, r5, r6, r7, r8, r9;
    int fp, sp, pc, sl;
    int f4[3], f5[3], f6[3], f7[3];
} _kernel_unwindblock;
```

Entry no. 4: `int _kernel_unwind(_kernel_unwindblock *inout, char **language)`

Unwinds the call stack one level. Returns:

>0 if it succeeds
 0 if it fails because it has reached the stack end or
 <0 if it fails for any other reason (e.g. stack corrupt)

Input values for *fp*, *sl* and *pc* must be correct. r4-r9 and f4-f7 are updated if the frame addressed by the input value of *fp* contains saved values for the corresponding registers.

fp, *sp*, *sl* and *pc* are always updated, the word pointed to by *language* is updated to point to a string naming the language corresponding to the returned value of *pc*.

Program environment functions

Entry no. 5: `char *_kernel_procname(int pc)`

Returns a string naming the procedure containing the address *pc* (or 0 if no name for it can be found).

Entry no. 6: `char *_kernel_language(int pc)`

Returns a string naming the language in whose code the address *pc* lies (or 0 if it is in no known language).

Entry no. 7: `char *_kernel_command_string(void)`

Returns a pointer to a copy of the command string used to run the program.

Entry no. 2: `void _kernel_setreturncode(unsigned code)`

Sets the return code to be used by `_kernel_exit`.

Entry no. 1: `void _kernel_exit(void)`

Calls `OS_Exit` with the return code specified by a previous call to `_kernel_setreturncode`.

Entry no. 47: void _kernel_raise_error(_kernel_oserror *)

Generates an external error.

Entry no. 3: void _kernel_exittraphandler(void)

Resets the InTrapHandler flag which prevents recursive traps. Used in trap handlers which do not return directly but continue execution. For example, the longjmp function in the C library calls _kernel_exittraphandler if called from within a signal handler.

Entry no. 8: int _kernel_hostos(void)

Returns 6 for RISC OS.
(Returns the result of calling OS_Byte with R0 = 0 and R1 = 1.)

Entry no. 37: int _kernel_fpavailable(void)

Returns non-zero if floating point is available.

Entry no. 21: _kernel_oserror *_kernel_last_oserror(void)

Returns a pointer to an error block describing the last OS error since _kernel_last_oserror was last called (or since the program started if there has been no such call). If there has been no OS error it returns 0. Note that occurrence of a further error may overwrite the contents of the block. This can be used, for example, to determine the error which caused fopen to fail. If _kernel_swi caused the last OS error, the error already returned by that call gets returned by this too.

Entry no. 23: _kernel_oserror *_kernel_getenv(const char *name, char *buffer, unsigned size)

Reads the value of a system variable, placing the value string in the buffer (of size size).

Entry no. 24: _kernel_oserror *_kernel_setenv(const char *name, const char *value)

Updates the value of a system variable to be string valued, with the given value (value = 0 deletes the variable).

Entry no. 43: int _kernel_escape_seen(void)

Returns 1 if there has been an escape since the previous call of _kernel_escape_seen (or since the program start if there has been no previous call). Escapes are never ignored with this mechanism, whereas they may be with the language EventProc mechanism since there may be no stack to call the EventProc on.

Entry no. 39: void _kernel_irqs_on(void)

Enable interrupts. You should not disable interrupts unless absolutely necessary. If you disable interrupts you should re-enable them as soon as possible (preferably within 10uS).

This function can only be used from code running in SVC mode.

Entry no. 40: void _kernel_irqs_off(void)

Disable IRQ interrupts. You should not disable interrupts unless absolutely necessary. If you disable interrupts you should re-enable them as soon as possible (preferably within 10uS).

This function can only be used from code running in SVC mode.

Entry no. 41: int _kernel_irqs_disabled(void)

Returns non-zero if IRQ interrupts are disabled.

General utility functions

```
typedef struct {
    int r[10]; /* only r0 - r9 matter for swi's */
} _kernel_swi_regs;
```

Entry no. 9: _kernel_oserror * _kernel_swi(int no, _kernel_swi_regs *in, _kernel_swi_regs *out)

Call the SWI specified by no. The X bit is set by _kernel_swi unless bit 31 of the SWI no (in no) is set. in and out are pointers to blocks for R0 - R9 on entry to and exit from the SWI.

Returns a pointer to an error block if an error occurred, otherwise 0.

Entry no. 45: _kernel_oserror * _kernel_swi_c(int no, _kernel_swi_regs *in, _kernel_swi_regs *out, int *carry)

Similar to _kernel_swi but returns the status of the carry flag on exit from the SWI in the word pointed to by carry.

Entry no. 10: `int _kernel_osbyte(int op, int x, int y)`

Performs an OS_Byte operation. If there is no error, the result contains:
the return value of R1 (x) in its bottom byte
the return value of R2 (y) in its second byte
1 in the third byte if carry is set on return, otherwise 0
0 in its top byte

Note that some OS_Byte calls return values too great to fit in a single byte.

Entry no. 11: `int _kernel_osrdch(void)`

Returns a character read from the currently selected OS input stream.

Entry no. 12: `int _kernel_oswrch(int ch)`

Writes a byte to all currently selected OS output streams. The return value just indicates success or failure.

Entry no. 13: `int _kernel_osbget(unsigned handle);`

Returns the next byte from the file identified by *handle*. (-1 ⇒ EOF)

Entry no. 14: `int _kernel_osbput(int ch, unsigned handle)`

Writes a byte to the file identified by *handle*. The return value just indicates success or failure.

```
typedef struct {
    void * dataptr; /* memory address of data */
    int nbytes, fileptr;
    int buf_len; /* these fields for RISC OS gppb extensions */
    char * wild_fld; /* points to wildcarded filename to match */
} _kernel_osgppb_block;
```

Entry no. 15: `int _kernel_osgppb(int op, unsigned handle, _kernel_osgppb_block *inout);`

Reads or writes a number of bytes from a filing system. The return value just indicates success or failure. Note that for some operations, the return value of C is significant, and for others it isn't. In all cases, therefore, a return value of -1 is possible, but for some operations it should be ignored.

Entry no. 16: `int _kernel_osword(int op, int *data)`

Performs an OS_Word operation. The size and format of the block pointed to by *data* depends on the particular OS_Word being used; it may be updated.

Entry no. 17: int _kernel_osfind(int op, char *name)

Opens or closes a file. Open returns a file handle (0 ⇒ open failed without error). For close the return value just indicates success or failure.

```
typedef struct {
    int load, exec; /* load, exec addresses */
    int start, end; /* start address/length, end address/attributes */
} _kernel_osfile_block;
```

Entry no. 18: int _kernel_osfile(int op, const char *name, _kernel_osfile_block *inout)

Performs an OS_File operation, with values of R2 - R5 taken from the osfile block. The block is updated with the return values of these registers, and the result is the return value of R0 (or an error indication).

Entry no. 19: int _kernel_osargs(int op, unsigned handle, int arg)

Performs an OS_Args operation. The result is the current filing system number (if op = 0) otherwise the value returned in R2 by the OS_Args operation.

Entry no. 20: int _kernel_oscli(char *s)

Calls OS_CLI with the specified string. If used to run another application the current application will be closed down. If you wish to return to the current application use _kernel_system. Any return value indicates an error in _kernel_oscli itself.

Entry no. 22: int _kernel_system(char *string, int chain)

Calls OS_CLI with the specified string. If chain is 0, the current application is copied to the top of memory first, then handlers are installed so that if the command string causes an application to be invoked, control returns to _kernel_system, which then copies the calling application back into its proper place. Hence the command is executed as a sub-program. If chain is 1, all handlers are removed before calling the CLI, and if it returns (the command is built-in) _kernel_system exits. Any return value indicates an error in _kernel_system itself.

Memory allocation functions

Entry no. 26: `unsigned _kernel_alloc(unsigned words, void **block)`

Tries to allocate a block of size = *words* words. Failing that, it allocates the largest possible block (may be size zero). If *words* is < 2048 it is rounded up to 2048. Returns a pointer to the allocated block in the word pointed to by *block*. The return value gives the size of the allocated block.

```
typedef void freeproc(void *);  
typedef void * allocproc(unsigned);
```

Entry no. 25: `void _kernel_register_allocs(allocproc *malloc, freeproc *free)`

Registers procedures to be used by the kernel when it requires to free or allocate storage. Currently this is only used to allocate and free stack chunks. Since `allocproc` and `freeproc` are called during stack extension, they must not check for stack overflow themselves or call any procedure which does stack checking and must guarantee to require no more than 41 words of stack.

The kernel provides default `alloc` and `free` procedures, however you should replace these with your own procedures since the default procedures are rather naive.

```
typedef int _kernel_ExtendProc(int /*n*/, void** /*p*/);
```

Entry no. 46: `_kernel_ExtendProc *_kernel_register_slotextend (_kernel_ExtendProc *proc)`

When the initial heap (supplied to `_kernel_init`) is full, the kernel is normally capable of extending it by extending the wimp slot. However, if the heap limit is not the same as the application limit, it is assumed that someone else has acquired the space between, and the procedure registered here is called to request *n* bytes from it.

Its return value is expected to be $\geq n$, or 0 to indicate failure. If successful the word pointed to by *p* should be set to the space allocated.

Language support functions

Entry no. 31: `unsigned _kernel_udiv(unsigned divisor, unsigned dividend);`

Divide and remainder function, returns the remainder in R1.

Entry no. 32: unsigned _kernel_urem(unsigned divisor, unsigned dividend);

Remainder function.

Entry no. 33: unsigned _kernel_udiv10(unsigned dividend);

Divide and remainder function, returns the remainder in R1.

Entry no. 34: int _kernel_sdiv(int divisor, int dividend);

Signed divide and remainder function, returns the remainder in R1.

Entry no. 35: int _kernel_srem(int divisor, int dividend);

Signed remainder function.

Entry no. 36: int _kernel_sdiv10(int dividend);

Signed divide and remainder function, returns the remainder in R1.

C library functions

The C library functions are grouped under the following headings:

- *Language support functions*
Provides functions for trap and event handling, initialisation and finalisation, and mathematical routines such as number conversion and multiplication.
- *assert*
The assert module provides one function which is useful during program testing.
- *ctype*
The ctype module provides several functions useful for testing and mapping characters.
- *errno*
The word variable `__errno` at offset 800 in the library statics is set whenever certain error conditions arises.
- *locale*
This module handles national characteristics, such as the different orderings month-day-year (USA) and day-month-year (UK).
- *math*
This module contains the prototypes for 22 mathematical functions. All return the type double.
- *setjmp*
This module provides two functions for bypassing the normal function call and return discipline.
- *signal*
Signal provides two functions.
- *stdio*
stdio provides many functions for performing input and output.
- *stdlib*
stdlib provides several general purpose functions.
- *string*
string provides several functions useful for manipulating character arrays and other objects treated as character arrays.
- *time*
time provides several functions for manipulating time.

Index of C library functions by entry number

entry no.	name	on page
0	trapHandler	page 4-298
1	uncaughtTrapHandler	page 4-298
2	eventHandler	page 4-299
3	unhandledEventHandler	page 4-299
4	x\$stack_overflow	page 4-300
5	x\$stack_overflow_1	page 4-300
6	x\$sdivide	page 4-300
7	x\$remainder	page 4-300
8	x\$divide	page 4-300
9	x\$divtest	page 4-300
10	x\$remainder	page 4-300
11	x\$multiply	page 4-300
12	_rd1chk	page 4-301
13	_rd2chk	page 4-301
14	_rd4chk	page 4-301
15	_wr1chk	page 4-301
16	_wr2chk	page 4-301
17	_wr4chk	page 4-301
18	_main	page 4-301
19	_exit	page 4-302
20	_clib_initialise	page 4-302
21	_backtrace	page 4-303
22	_count	page 4-303
23	_count1	page 4-303
24	_stfp	page 4-303
25	_ldfp	page 4-303
26	_printf	page 4-318
27	_fprintf	page 4-319
28	_sprintf	page 4-319
29	clock	page 4-340
30	difftime	page 4-340
31	mktime	page 4-340
32	time	page 4-341
33	asctime	page 4-341
34	ctime	page 4-341
35	gmtime	page 4-341
36	localtime	page 4-341
37	strftime	page 4-341
38	memcpy	page 4-335
39	memmove	page 4-335

entry no.	name	on page
40	strcpy	page 4-335
41	strncpy	page 4-335
42	strcat	page 4-336
43	strncat	page 4-336
44	memcmp	page 4-336
45	strcmp	page 4-336
46	strncmp	page 4-336
47	memchr	page 4-337
48	strchr	page 4-337
49	strcspn	page 4-337
50	strpbrk	page 4-338
51	strchr	page 4-338
52	strspn	page 4-338
53	strstr	page 4-338
54	strtok	page 4-338
55	memset	page 4-339
56	strerror	page 4-339
57	strlen	page 4-339
58	atof	page 4-327
59	atoi	page 4-327
60	atol	page 4-327
61	strtod	page 4-327
62	strtol	page 4-327
63	strtoul	page 4-328
64	rand	page 4-329
65	srand	page 4-329
66	calloc	page 4-329
67	free	page 4-329
68	malloc	page 4-329
69	realloc	page 4-329
70	abort	page 4-330
71	atexit	page 4-330
72	exit	page 4-330
73	getenv	page 4-330
74	system	page 4-331
75	bsearch	page 4-331
76	qsort	page 4-331
77	abs	page 4-332
78	div	page 4-332
79	labs	page 4-332

entry no.	name	on page
80	ldiv	page 4-332
81	remove	page 4-314
82	rename	page 4-314
83	tmpfile	page 4-314
84	_old_tmpnam	page 4-315
85	fclose	page 4-315
86	fflush	page 4-315
87	fopen	page 4-315
88	freopen	page 4-316
89	setbuf	page 4-316
90	setvbuf	page 4-317
91	printf	page 4-318
92	fprintf	page 4-317
93	sprintf	page 4-318
94	scanf	page 4-320
95	fscanf	page 4-319
96	sscanf	page 4-320
97	vprintf	page 4-320
98	vfprintf	page 4-321
99	vsprintf	page 4-321
100	_vprintf	page 4-319
101	fgetc	page 4-321
102	fgets	page 4-321
103	fputc	page 4-321
104	fputs	page 4-322
105	__filbuf	page 4-326
106	getc	page 4-322
107	getchar	page 4-322
108	gets	page 4-322
109	__flsbuf	page 4-326
110	putc	page 4-322
111	putchar	page 4-323
112	puts	page 4-323
113	ungetc	page 4-323
114	fread	page 4-323
115	fwrite	page 4-324
116	fgetpos	page 4-324
117	fseek	page 4-324
118	fsetpos	page 4-325
119	ftell	page 4-325
120	rewind	page 4-325
121	clearerr	page 4-325

entry no.	name	on page
122	feof	page 4-325
123	ferror	page 4-326
124	perror	page 4-326
125	__ignore_signal_handler	page 4-313
126	__error_signal_marker	page 4-313
127	__default_signal_handler	page 4-313
128	signal	page 4-311
129	raise	page 4-312
130	setjmp	page 4-311
131	longjmp	page 4-311
132	acos	page 4-309
133	asin	page 4-309
134	atan	page 4-309
135	atan2	page 4-309
136	cos	page 4-309
137	sin	page 4-309
138	tan	page 4-309
139	cosh	page 4-309
140	sinh	page 4-309
141	tanh	page 4-309
142	exp	page 4-309
143	frexp	page 4-310
144	ldexp	page 4-310
145	log	page 4-310
146	log10	page 4-310
147	modf	page 4-310
148	pow	page 4-310
149	sqrt	page 4-310
150	ceil	page 4-310
151	fabs	page 4-310
152	floor	page 4-310
153	fmod	page 4-310
154	setlocale	page 4-308
155	isalnum	page 4-305
156	isalpha	page 4-305
157	iscntrl	page 4-305
158	isdigit	page 4-305
159	isgraph	page 4-305
160	islower	page 4-305
161	isprint	page 4-306
162	ispunct	page 4-306
163	isspace	page 4-306

entry no.	name	on page
164	isupper	page 4-306
165	isxdigit	page 4-306
166	tolower	page 4-306
167	toupper	page 4-306
168	__assert	page 4-305
169	_memcpy	page 4-304
170	_memset	page 4-304
171	localeconv	page 4-308
172	mblen	page 4-333
173	mbtowc	page 4-333
174	wctomb	page 4-333
175	mbstowcs	page 4-334
176	wcstombs	page 4-334
177	strxfrm	page 4-337
178	strcoll	page 4-336
179	_clib_finalisemodule	page 4-304
180	_clib_version	page 4-304
181	finalise	page 4-304
182	tmpnam	page 4-314
error condition	EDOM	page 4-307
error condition	ERANGE	page 4-307
error condition	ESIGNUM	page 4-307

Index of C library functions by function name

name	entry no.	on page
abort	70	page 4-330
abs	77	page 4-332
acos	132	page 4-309
asctime	33	page 4-341
asin	133	page 4-309
__assert	168	page 4-305
atan	134	page 4-309
atan2	135	page 4-309
atexit	71	page 4-330
atof	58	page 4-327
atoi	59	page 4-327
atol	60	page 4-327
_backtrace	21	page 4-303
bsearch	75	page 4-331
calloc	66	page 4-329
ceil	150	page 4-310

name	entry no.	on page
clearerr	121	page 4-325
_clib_finalisemodule	179	page 4-304
_clib_initialise	20	page 4-302
_clib_version	180	page 4-304
clock	29	page 4-340
cos	136	page 4-309
cosh	139	page 4-309
_count	22	page 4-303
_count1	23	page 4-303
ctime	34	page 4-341
__default_signal_handler	127	page 4-313
difftime	30	page 4-340
div	78	page 4-332
__error_signal_marker	126	page 4-313
eventHandler	2	page 4-299
exit	72	page 4-330
_exit	19	page 4-302
exp	142	page 4-309
fabs	151	page 4-310
fclose	85	page 4-315
feof	122	page 4-325
ferror	123	page 4-326
fflush	86	page 4-315
fgetc	101	page 4-321
fgetpos	116	page 4-324
fgets	102	page 4-321
__filbuf	105	page 4-326
finalise	181	page 4-304
floor	152	page 4-310
__flsbuf	109	page 4-326
fmod	153	page 4-310
fopen	87	page 4-315
fprintf	92	page 4-317
_fprintf	27	page 4-319
fputc	103	page 4-321
fputs	104	page 4-322
fread	114	page 4-323
free	67	page 4-329
freopen	88	page 4-316
frexp	143	page 4-310
fscanf	95	page 4-319
fseek	117	page 4-324

name	entry no.	on page
fsetpos	118	page 4-325
ftell	119	page 4-325
fwrite	115	page 4-324
getc	106	page 4-322
getchar	107	page 4-322
getenv	73	page 4-330
gets	108	page 4-322
gmtime	35	page 4-341
__ignore_signal_handler	125	page 4-313
isalnum	155	page 4-305
isalpha	156	page 4-305
iscntrl	157	page 4-305
isdigit	158	page 4-305
isgraph	159	page 4-305
islower	160	page 4-305
isprint	161	page 4-306
ispunct	162	page 4-306
isspace	163	page 4-306
isupper	164	page 4-306
isxdigit	165	page 4-306
labs	79	page 4-332
localeconv	171	page 4-308
ldexp	144	page 4-310
_ldfp	25	page 4-303
ldiv	80	page 4-332
localtime	36	page 4-341
log	145	page 4-310
log10	146	page 4-310
longjmp	131	page 4-311
_main	18	page 4-301
malloc	68	page 4-329
mblen	172	page 4-333
mbstowcs	175	page 4-334
mbtowc	173	page 4-333
memchr	47	page 4-337
memcmp	44	page 4-336
memcpy	38	page 4-335
_memcpy	169	page 4-304
memmove	39	page 4-335
memset	55	page 4-339
_memset	170	page 4-304
mktime	31	page 4-340

name	entry no.	on page
modf	147	page 4-310
_old_tmpnam	84	page 4-315
perror	124	page 4-326
pow	148	page 4-310
printf	91	page 4-318
_printf	26	page 4-318
putc	110	page 4-322
putchar	111	page 4-323
puts	112	page 4-323
qsort	76	page 4-331
raise	129	page 4-312
rand	64	page 4-329
_rd1chk	12	page 4-301
_rd2chk	13	page 4-301
_rd4chk	14	page 4-301
realloc	69	page 4-329
remove	81	page 4-314
rename	82	page 4-314
rewind	120	page 4-325
scanf	94	page 4-320
setbuf	89	page 4-316
setjmp	130	page 4-311
setlocale	154	page 4-308
setvbuf	90	page 4-317
signal	128	page 4-311
sin	137	page 4-309
sinh	140	page 4-309
sprintf	93	page 4-318
_sprintf	28	page 4-319
sqrt	149	page 4-310
srand	65	page 4-329
sscanf	96	page 4-320
_stfp	24	page 4-303
strcat	42	page 4-336
strchr	48	page 4-337
strcmp	45	page 4-336
strcoll	178	page 4-336
strcpy	40	page 4-335
strcspn	4	page 4-337
strerror	56	page 4-339
strftime	37	page 4-341
strlen	57	page 4-339

name	entry no.	on page
strncat	43	page 4-336
strncmp	46	page 4-336
strncpy	41	page 4-335
strpbrk	50	page 4-338
strchr	51	page 4-338
strspn	52	page 4-338
strstr	53	page 4-338
strtod	61	page 4-327
strtok	54	page 4-338
strtol	62	page 4-327
strtoul	63	page 4-328
strxfrm	177	page 4-337
system	74	page 4-331
tan	138	page 4-309
tanh	141	page 4-309
time	32	page 4-341
tmpfile	83	page 4-314
tmpnam	182	page 4-314
tolower	166	page 4-306
toupper	167	page 4-306
trapHandler	0	page 4-298
uncaughtTrapHandler	1	page 4-298
ungetc	113	page 4-323
unhandledEventHandler	3	page 4-299
vfprintf	98	page 4-321
vprintf	97	page 4-320
_vprintf	100	page 4-319
vsprintf	99	page 4-321
wcstombs	176	page 4-334
wctomb	174	page 4-333
_wr1chk	15	page 4-301
_wr2chk	16	page 4-301
_wr4chk	17	page 4-301
x\$divide	8	page 4-300
x\$divtest	9	page 4-300
x\$multiply	11	page 4-300
x\$remainder	10	page 4-300
x\$stack_overflow	4	page 4-300
x\$stack_overflow_1	5	page 4-300
x\$divide	6	page 4-300
x\$remainder	7	page 4-300

Language support functions

Entry no. 0: TrapHandler

Entry no. 1: UncaughtTrapHandler

On entry:

R0 = error code

R1 = pointer to register dump

On exit:

Only exits if the trap was not handled

R0 = 0 (indicating that the trap was not handled).

These are the default TrapProc and UncaughtTrapProc handlers used by the C library in its kernel language description (see the section entitled *Interfacing a language run-time system to the Acorn library kernel* on page 4-242).

You may use these entries in your own kernel language description if you wish to have trap handling similar to that provided by the C library, or you may call these entries directly from your own trap handler if you wish to perform some pre-processing before passing the trap on.

The error code on entry is converted to a signal number as follows:

Signal no.	Error codes
2 (SIGFPE)	&8000020 (Error_DivideByZero), &80000200 (Error_FPBase) – &800002FF (Error_FPLimit – 1)
3 (SIGILL)	&8000000 (Error_IllegalInstruction), &80000001 (Error_PrefetchAbort), &80000005 (Error_BranchThroughZero)
5 (SIGSEGV)	&80000002 (Error_DataAbort), &80000003 (Error_AddressException), &80800EA0 (Error_ReadFail), &80800EA1 (Error_WriteFail)
7 (SIGSTAK)	&80000021 (Error_StackOverflow)
10 (SIGOSERROR)	All other errors

It then determines whether a signal handler has been set up for the converted signal handler; if no such handler has been set up (ie the signal handler is set to `__SIG_DFL`) it returns with R0 = 0.

Otherwise it calls the C library function `raise` with the derived signal number. If the `raise` function returns (ie the signal handler returns) a postmortem stack backtrace is generated.

Entry no. 2: EventHandler

Entry no. 3: UnhandledEventHandler

On entry:

R0 = event code

R1 = pointer to register dump

On exit:

R0 = 1 if the event was handled, else 0

These are the default `EventProc` and `UnhandledEventProc` handlers used by the C library in its kernel language description (see the section entitled *Interfacing a language run-time system to the Acorn library kernel* on page 4-242).

You may use these entries in your own kernel language description if you wish to have event handling similar to that provided by the C library or you may call these entries directly from your own event handler if you wish to perform some pre-processing before passing the event on.

The event code on entry is either a RISC OS event number as described in the chapter entitled *Events* on page 1-147, or `-1` to indicate an escape event.

All events codes except `-1` are currently ignored. The handler simply returns with `R0 = 0` if `R0 ≠ -1` on entry.

`EventHandler` then determines whether a `SIGINT` signal handler has been set up. If no handler is set up (ie the signal handler is set to `__SIG_DFL`) `EventHandler` returns with `R0 = 0`.

The C library function `raise` is then called with the signal number `SIGINT`. Note: `raise` is always called by `UnhandledEventHandler` even if the signal handler is set to `__SIG_DFL`.

If the signal handler returns the event handler returns with `R0 = 1`.

Certain sections of the C library are non-reentrant. When these sections are entered they set the variable `_interrupts_off` at offset 964 in the library statics to 1.

`EventHandler` and `UnhandledEventHandler` check this variable and, if it is set, they set the variable `_saved_interrupt` at offset 968 in the library statics to `SIGINT` and return immediately with `R0 = 1` and without calling `raise`.

When the non-reentrant sections of code finish they reset the variable `_interrupts_off` and check the variable `_saved_interrupts`. If `_saved_interrupts` is non-zero it is reset to zero and the signal number stored in `_saved_interrupts` (before it was reset to 0) is raised.

Entry no. 4: `x$stack_overflow`

This entry branches directly to `_kernel_stkovf_split_0frame` which is described in the section entitled *How the run-time stack is managed and extended* on page 4-243.

Entry no. 5: `x$stack_overflow_1`

This entry branches directly to `_kernel_stkovf_split` which is described in the section entitled *How the run-time stack is managed and extended* on page 4-243.

Entry no. 6: `x$udivide`

This entry branches directly to `_kernel_udiv` described on page 4-286.

Entry no. 7: `x$uremainder`

This entry branches directly to `_kernel_urem` described on page 4-287.

Entry no. 8: `x$divide`

This entry branches directly to `_kernel_sdiv` described on page 4-287.

Entry no. 9: `x$divtest`

This function is used by the C compiler to test for division by zero when the result of the division is discarded.

If R0 is non-zero the function simply returns. Otherwise it generates a Divide by zero error.

Entry no. 10: `x$remainder`

This entry branches directly to `_kernel_srem` described on page 4-287.

Entry no. 11: `x$multiply`

On entry:

R0 = multiplicand

R1 = multiplier

On exit:

R0 = R0 × R1

R1, R2 scrambled.

Entry no. 12: `_rd1chk`**Entry no. 13: `_rd2chk`****Entry no. 14: `_rd4chk`**

The functions `_rd1chk`, `_rd2chk` and `_rd4chk` check that the value of R0 passed to them is a valid address in the application space ($&8000 \leq R0 < &1000000$). `_rd2chk` and `_rd4chk` also check that the value is properly aligned for a half-word / word access respectively.

If the value of R0 is a valid address the function just returns, otherwise it generates an Illegal read error.

These calls are used by the C compiler when compiling with memory checking enabled.

Entry no. 15: `_wr1chk`**Entry no. 16: `_wr2chk`****Entry no. 17: `_wr4chk`**

The functions `_wr1chk`, `_wr2chk` and `_wr4chk` check that the value of R0 passed to them is a valid address in the application space ($&8000 \leq R0 < &1000000$). `_rd2chk` and `_rd4chk` also check that the value is properly aligned for a half-word / word access respectively.

If the value of R0 is a valid address the function just returns, otherwise it generates an Illegal write error.

These calls are used by the C compiler when compiling with memory checking enabled.

Entry no. 18: `_main`**On entry:**

R0 = pointer to copy of command line (the command line pointed to by R0 on return from `OS_GetEnv` should be copied to another buffer before calling `_main`; this can be done using `_kernel_command_string`, detailed on page 4-281).

R1 = address of routine at which execution will continue when `_main` has finished.

The following entry and exit conditions apply for this routine:

On entry:

R0 = count of argument words.

R1 = pointer to block containing $R0 + n$ words, each word of which points to a zero terminated string which is the n th word in the command line passed to `_main`. The last word in the block contains 0.

On exit:

R0 = exit condition (0 = success, else failure)

For C programs this argument will generally point at main.

On exit:

Does not return. Control is regained through the R1 argument on entry.

This function parses the command line pointed to by R0 and then calls the function pointed to by R1.

For C programs this function is called by the C library as a precursor to calling main to provide the C entry / exit requirements.

Entry no. 19: void `_exit(void)`

This function is identical in behaviour to the C library function `exit` described on page 4-330.

Entry no. 20: void `_clib_initialise(void)`

Performs initialisation required by the C library before other C library functions can be called. You may call kernel library functions without first making this call. You should call this function in your initialisation entry for a module and in your `InitProc` procedure for applications or modules that have a run entry. For a description of `InitProc` procedures, see page 4-257. The two programming examples on page 4-264 and page 4-266 show how `_clib_initialise` should be called for an application and a module respectively.

Entry no. 21: void _backtrace(int why, int *address, _kernel_unwindblock *uwb)

Displays a stack backtrace and exits with the exit code 1.

The `_kernel_unwindblock` structure is described with the `_kernel_unwind` function on page 4-281. The argument `why` is an error code, if `why` is `Error_ReadFail` (&80800ea0) or `Error_WriteFail` (&80800ea1) the address given by the `address` argument is displayed at the top of the backtrace, otherwise the message `postmortem` requested is displayed.

Entry no. 22: _count**Entry no. 23: _count1**

These entries are used by the C compiler when generating *profile* code.

Both `_count` and `_count1` increment the word pointed to by R14 (after stripping the status bits); this will generally be the word immediately following a BL instruction to the relevant routine. `_count` then returns to the word immediately following the incremented word, `_count1` returns to the word after that (the second word is used by the C compiler to record the position in a source file that this count-point refers to).

```
BL      _count
DCD     0          ; This word incremented each time _count is called
...     ; Control returns here

BL      _count1
DCD     0          ; This word incremented each time _count1 is called
DCD     filepos   ; Offset into source file
...     ; Control returns here
```

Entry no. 24: void _stfp(double d, void *x)

This function converts the double FP no. `d` to packed decimal and stores it at address `x`. Note that the double `d` is passed in R0, R1 (R0 containing the first word when a double is stored in memory, R1 containing the second word), the argument `x` is passed in R2. Three words should be reserved at `x` for the packed decimal number.

Entry no. 25: double _ldfp(void *x)

This function converts the packed decimal number stored at `x` to a double FP no. and returns this in F0.

Entry no. 169: void _memcpy(int *dest, int *source, int n)

This function performs a similar function to memcpy except that dest and source must be word aligned and the byte count n must be a multiple of 4.

It is used by the C compiler when copying structures.

Entry no. 170: void _memset(int *dest, int w, int n)

This function performs a similar function to memset except that dest must be word aligned, the byte value to be set must be copied into each of the four bytes of w (i.e. to initialise memory to &01 you must use &01010101 in w) and the byte count n must be a multiple of 4.

It is used by the C compiler when initialising structures.

Entry no. 179: _clib_finalisemodule

On entry:

R0 = private word pointer

On exit:

Block pointed to by private word is freed

This entry must be called in the finalisation code of a module which uses the shared C library. Before calling it you must set up the static data relocation pointers on the base of the SVC stack and initialise the SL register to point to the base of the SVC stack + 512. The old static data relocation pointers on the base of the SVC stack must be saved around this call.

Entry no. 180: char *_clib_version(void)

This function returns a string giving version information on the shared C library.

Entry no. 181: Finalise

This function calls all the registered atexit functions and then performs some internal finalisation of the alloc and io subsystems.

This entry is called automatically by the C library on finalisation; you should not call it in your code.

assert

The assert module provides one function which is useful during program testing.

Entry no. 168: void `__assert(char *reason, char *file, int line)`

Displays the message:

```
*** assertion failed: 'reason', file 'file', line 'line'
```

and raises SIGABRT.

This function is generally used within a macro which calls `__assert` if a specified condition is false.

ctype

The ctype module provides several functions useful for testing and mapping characters. In all cases the argument is an int, the value of which is representable as an unsigned char or equal to the value `-1`. If the argument has any other value, the behaviour is undefined.

Entry no. 155: int `isalnum(int c)`

Returns true if `c` is alphabetic or numeric

Entry no. 156: int `isalpha(int c)`

Returns true if `c` is alphabetic

Entry no. 157: int `iscntrl(int c)`

Returns true if `c` is a control character (in the ASCII locale)

Entry no. 158: int `isdigit(int c)`

Returns true if `c` is a decimal digit

Entry no. 159: int `isgraph(int c)`

Returns true if `c` is any printable character other than space

Entry no. 160: int `islower(int c)`

Returns true if `c` is a lower-case letter

Entry no. 161: int isprint(int c)

Returns true if *c* is a printable character (in the ASCII locale this means &20 (space) → &7E (tilde) inclusive).

Entry no. 162: int ispunct(int c)

Returns true if *c* is a printable character other than a space or alphanumeric character

Entry no. 163: int isspace(int c)

Returns true if *c* is a white space character viz: space, newline, return, linefeed, tab or vertical tab

Entry no. 164: int isupper(int c)

Returns true if *c* is an upper-case letter

Entry no. 165: int isxdigit(int c)

Returns true if *c* is a hexadecimal digit, ie in 0...9, a...f, or A...F

Entry no. 166: int tolower(int c)

Forces *c* to lower case if it is an upper-case letter, otherwise returns the original value

Entry no. 167: int toupper(int c)

Forces *c* to upper case if it is a lower-case letter, otherwise returns the original value

errno

The word variable `errno` at offset 800 in the library statics is set whenever one of the error conditions listed below arises.

EDOM (errno=1)

If a domain error occurs (an input argument is outside the domain over which the mathematical function is defined) the integer expression `errno` acquires the value of the macro `EDOM`, and `HUGE_VAL` is returned. `EDOM` may be used by non-mathematical functions.

ERANGE (errno=2)

A range error occurs if the result of a function cannot be represented as a double value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro `HUGE_VAL`, with the same sign as the correct value of the function; the integer expression `errno` acquires the value of the macro `ERANGE`. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero; the integer expression `errno` acquires the value of the macro `ERANGE`. `ERANGE` may be used by non-mathematical functions.

ESIGNAL (errno=3)

If an unrecognised signal is caught by the default signal handler, `errno` is set to `ESIGNAL`.

locale

This module handles national characteristics, such as the different orderings month-day-year (USA) and day-month-year (UK).

Entry no. 154: `char *setlocale(int category, const char *locale)`

Selects the appropriate part of the program's locale as specified by the category and locale arguments. The `setlocale` function may be used to change or query the program's entire current locale or portions thereof. Locale information is divided into the following types:

Type	Value	Description
LC_COLLATE	(1)	string collation
LC_CTYPE	(2)	character type
LC_MONETARY	(4)	monetary formatting
LC_NUMERIC	(8)	numeric string formatting
LC_TIME	(16)	time formatting
LC_ALL	(31)	entire locale

The locale string specifies which locale set of information is to be used. For example,

```
setlocale(LC_MONETARY,"uk")
```

would insert monetary information into the `lconv` structure. To query the current locale information, set the locale string to null and read the string returned.

Entry no. 171: `struct lconv *localeconv(void)`

Sets the components of an object with type `struct lconv` with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale. The members of the structure with type `char *` are strings, any of which (except `decimal_point`) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type `char` are non-negative numbers, any of which can be `CHAR_MAX` to indicate that the value is not available in the current locale. The members included are described above.

`localeconv` returns a pointer to the filled in object. The structure pointed to by the return value will not be modified by the program, but may be overwritten by a subsequent call to the `localeconv` function. In addition, calls to the `setlocale` function with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may overwrite the contents of the structure.

math

This module contains 22 mathematical functions. All return the type `double`.

Entry no. 132: double acos(double x)

Returns arc cosine of x . A domain error occurs for arguments not in the range -1 to 1

Entry no. 133: double asin(double x)

Returns arc sine of x . A domain error occurs for arguments not in the range -1 to 1

Entry no. 134: double atan(double x)

Returns arc tangent of x

Entry no. 135: double atan2(double x, double y)

Returns arc tangent of x/y

Entry no. 136: double cos(double x)

Returns cosine of x (measured in radians)

Entry no. 137: double sin(double x)

Returns sine of x (measured in radians)

Entry no. 138: double tan(double x)

Returns tangent of x (measured in radians)

Entry no. 139: double cosh(double x)

Returns hyperbolic cosine of x

Entry no. 140: double sinh(double x)

Returns hyperbolic sine of x

Entry no. 141: double tanh(double x)

Returns hyperbolic tangent of x

Entry no. 142: double exp(double x)

Returns exponential function of x

Entry no. 143: double frexp(double x, int *exp)

Returns the value x , such that x is a double with magnitude in the interval 0.5 to 1.0 or zero, and value equals x times 2 raised to the power $*exp$

Entry no. 144: double ldexp(double x, int exp)

Returns x times 2 raised to the power of exp

Entry no. 145: double log(double x)

Returns natural logarithm of x

Entry no. 146: double log10(double x)

Returns log to the base 10 of x

Entry no. 147: double modf(double x, double *iptr)

Returns signed fractional part of x . Stores integer part of x in object pointed to by $iptr$.

Entry no. 148: double pow(double x, double y)

Returns x raised to the power of y

Entry no. 149: double sqrt(double x)

Returns positive square root of x

Entry no. 150: double ceil(double x)

Returns smallest integer not less than x (ie rounding up)

Entry no. 151: double fabs(double x)

Returns absolute value of x

Entry no. 152: double floor(double x)

Returns largest integer not greater than x (ie rounding down)

Entry no. 153: double fmod(double x, double y)

Returns floating-point remainder of x/y

setjmp

This module provides two functions for bypassing the normal function call and return discipline (useful for dealing with unusual conditions encountered in a low-level function of a program).

Entry no. 130: `int setjmp(jmp_buf env)`

The calling environment is saved in *env*, for later use by the `longjmp` function. If the return is from a direct invocation, the `setjmp` function returns the value zero. If the return is from a call to the `longjmp` function, the `setjmp` function returns a non-zero value.

Entry no. 131: `void longjmp(jmp_buf env, int val)`

The environment saved in *env* by the most recent call to `setjmp` is restored. If there has been no such call, or if the function containing the call to `setjmp` has terminated execution (eg with a return statement) in the interim, the behaviour is undefined. All accessible objects have values as at the time `longjmp` was called, except that the values of objects of automatic storage duration that do not have volatile type and that have been changed between the `setjmp` and `longjmp` calls are indeterminate.

As it bypasses the usual function call and return mechanism, the `longjmp` function executes correctly in contexts of interrupts, signals and any of their associated functions. However, if the `longjmp` function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behaviour is undefined.

After `longjmp` is completed, program execution continues as if the corresponding call to `setjmp` had just returned the value specified by *val*. The `longjmp` function cannot cause `setjmp` to return the value 0; if *val* is 0, `setjmp` returns the value 1.

signal

Signal provides two functions.

```
typedef void Handler(int);
```

Entry no. 128: `Handler *signal(int, Handler *)`;

The following signal handlers are defined:

Type	value	description
SIG_DFL	(Handler*)-1	default routine
SIG_IGN	(Handler*)-2	ignore signal routine
SIG_ERR	(Handler*)-3	dummy routine to flag error return from signal

The following signals are defined:

Signal	value	description
SIGABRT	1	abort (ie call to abort())
SIGFPE	2	arithmetic exception
SIGILL	3	illegal instruction
SIGINT	4	attention request from user
SIGSEGV	5	bad memory access
SIGTERM	6	termination request
SIGSTAK	7	stack overflow
SIGUSR1	8	user definable
SIGUSR2	9	user definable
SIGOSERROR	10	operating system error

The 'signal' function chooses one of three ways in which receipt of the signal number sig is to be subsequently handled. If the value of func is SIG_DFL, default handling for that signal will occur. If the value of func is SIG_IGN, the signal will be ignored. Otherwise func points to a function to be called when that signal occurs.

When a signal occurs, if func points to a function, first the equivalent of signal(sig, SIG_DFL) is executed. (If the value of sig is SIGILL, whether the reset to SIG_DFL occurs is implementation-defined (under RISC OS the reset does occur)). Next, the equivalent of (*func)(sig); is executed. The function may terminate by calling the abort, exit or longjmp function. If func executes a return statement and the value of sig was SIGFPE or any other implementation-defined value corresponding to a computational exception, the behaviour is undefined. Otherwise, the program will resume execution at the point it was interrupted.

If the signal occurs other than as a result of calling the abort or raise function, the behaviour is undefined if the signal handler calls any function in the standard library other than the signal function itself or refers to any object with static storage duration other than by assigning a value to a volatile static variable of type sig_atomic_t. At program start-up, the equivalent of signal(sig, SIG_IGN) may be executed for some signals selected in an implementation defined manner (under RISC OS this does not occur); the equivalent of signal(sig, SIG_DFL) is executed for all other signals defined by the implementation.

If the request can be honoured, the signal function returns the value of func for most recent call to signal for the specified signal sig. Otherwise, a value of SIG_ERR is returned and the integer expression errno is set to indicate the error.

Entry no. 129: int raise(int sig)

Sends the signal sig to the executing program. Returns zero if successful, non-zero if unsuccessful.

Entry no. 125: void __ignore_signal_handler(int sig)

This function is for compatibility with older versions of the shared C library stubs and should not be called in your code.

Entry no. 126: void __error_signal_marker(int sig)

This function is for compatibility with older versions of the shared C library stubs and should not be called in your code.

Entry no. 127: void __default_signal_handler(int sig)

This function is for compatibility with older versions of the shared C library stubs and should not be called in your code.

stdio

stdio provides many functions for performing input and output. For a discussion on Streams and Files refer to sections 4.9.2 and 4.9.3 in the ANSI standard.

The following two types are used by the stdio module:

```
typedef int fpos_t;
```

fpos_t is an object capable of recording all information needed to specify uniquely every position within a file.

```
typedef struct FILE{
    unsigned char *_ptr; /* pointer to IO buffer */
    int _icnt;          /* character count for input */
    int _ocnt;          /* character count for output */
    int _flag;          /* flags, see below */
    int internal[6];
}FILE;
```

The following flags are defined in the flags field above:

Flag	Bit mask	Description
_IOEOF	&040	end-of-file reached
_IOERR	&080	error occurred on stream
_IOFBF	&100	fully buffered IO
_IOLBF	&200	line buffered IO
_IONBF	&400	unbuffered IO

FILE is an object capable of recording all information needed to control a stream, such as its file position indicator, a pointer to its associated buffer, an error indicator that records whether a read/write error has occurred and an end-of-file indicator that records whether the end-of-file has been reached.

Entry no. 81: `int remove(const char *filename)`

Causes the file whose name is the string pointed to by *filename* to be removed. Subsequent attempts to open the file will fail, unless it is created anew. If the file is open, the behaviour of the remove function is implementation-defined (under RISC OS the operation fails).

Returns: zero if the operation succeeds, non-zero if it fails.

Entry no. 82: `int rename(const char *old, const char *new)`

Causes the file whose name is the string pointed to by *old* to be henceforth known by the name given by the string pointed to by *new*. The file named *old* is effectively removed. If a file named by the string pointed to by *new* exists prior to the call of the rename function, the behaviour is implementation-defined (under RISC OS, the operation fails).

Returns: zero if the operation succeeds, non-zero if it fails, in which case if the file existed previously it is still known by its original name.

Entry no. 83: `FILE *tmpfile(void)`

Creates a temporary binary file that will be automatically removed when it is closed or at program termination. The file is opened for update.

Returns: a pointer to the stream of the file that it created. If the file cannot be created, a null pointer is returned.

Entry no. 182: `char *tmpnam(char *s)`

Generates a string that is not the same as the name of an existing file. The `tmpnam` function generates a different string each time it is called, up to `TMP_MAX` times. If it is called more than `TMP_MAX` times, the behaviour is implementation-defined (under RISC OS the algorithm for the name generation works just as well after `tmpnam` has been called more than `TMP_MAX` times as before; a name clash is impossible in any single half year period).

Returns: If the argument is a null pointer, the `tmpnam` function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to the `tmpnam` function may modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least `L_tmpnam` characters; the `tmpnam` function writes its result in that array and returns the argument as its value.

Entry no. 84: `char *__old_tmpnam(char *s)`

This function is included for backwards compatibility for binaries linked with older library stubs. You should not call this function in your code, call `tmpnam` (Entry no. 182) instead.

Entry no. 85: `int fclose(FILE *stream)`

Causes the stream pointed to by *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated.

Returns: zero if the stream was successfully closed, or EOF if any errors were detected or if the stream was already closed.

Entry no. 86: `int fflush(FILE *stream)`

If the stream points to an output or update stream in which the most recent operation was output, the `fflush` function causes any unwritten data for that stream to be delivered to the host environment to be written to the file. If the stream points to an input or update stream, the `fflush` function undoes the effect of any preceding `ungetc` operation on the stream.

Returns: EOF if a write error occurs.

Entry no. 87: `FILE *fopen(const char *filename, const char *mode)`

Opens the file whose name is the string pointed to by *filename*, and associates a stream with it. The argument *mode* points to a string beginning with one of the following sequences:

<code>r</code>	open text file for reading
<code>w</code>	create text file for writing, or truncate to zero length
<code>a</code>	append; open text file or create for writing at eof
<code>rb</code>	open binary file for reading
<code>wb</code>	create binary file for writing, or truncate to zero length
<code>ab</code>	append; open binary file or create for writing at eof
<code>r+</code>	open text file for update (reading and writing)
<code>w+</code>	create text file for update, or truncate to zero length
<code>a+</code>	append; open text file or create for update, writing at eof
<code>r+b</code> or <code>rb+</code>	open binary file for update (reading and writing)
<code>w+b</code> or <code>wb+</code>	create binary file for update, or truncate to zero length
<code>a+b</code> or <code>ab+</code>	append; open binary file or create for update, writing at eof

- Opening a file with read mode (r as the first character in the *mode* argument) fails if the file does not exist or cannot be read.
- Opening a file with append mode (a as the first character in the *mode* argument) causes all subsequent writes to be forced to the current end of file, regardless of intervening calls to the *fseek* function.
- In some implementations, opening a binary file with append mode (b as the second or third character in the *mode* argument) may initially position the file position indicator beyond the last data written, because of null padding (but not under RISC OS).
- When a file is opened with update mode (+ as the second or third character in the *mode* argument), both input and output may be performed on the associated stream. However, output may not be directly followed by input without an intervening call to the *fflush* function or to a file positioning function (*fseek*, *fsetpos*, or *rewind*), nor may input be directly followed by output without an intervening call to the *fflush* function or to a file positioning function, unless the input operation encounters end-of-file.
- Opening a file with update mode may open or create a binary stream in some implementations (but not under RISC OS). When opened, a stream is fully buffered if and only if it does not refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Returns: a pointer to the object controlling the stream. If the open operation fails, *fopen* returns a null pointer.

Entry no. 88: FILE **freopen*(const char **filename*, const char **mode*, FILE **stream*)

Opens the file whose name is the string pointed to by *filename* and associates the stream pointed to by *stream* with it. The *mode* argument is used just as in the *fopen* function. The *freopen* function first attempts to close any file that is associated with the specified stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.

Returns: a null pointer if the operation fails. Otherwise, *freopen* returns the value of the stream.

Entry no. 89: void *setbuf*(FILE **stream*, char **buf*)

Except that it returns no value, the *setbuf* function is equivalent to the *setvbuf* function invoked with the values `_IOFBF` for *mode* and `BUFSIZ` for *size*, or if *buf* is a null pointer, with the value `_IONBF` for *mode*.

Returns: no value.

Entry no. 90: int setvbuf(FILE **stream*, char **buf*, int *mode*, size_t *size*)

This may be used after the stream pointed to by *stream* has been associated with an open file but before it is read or written. The argument *mode* determines how *stream* will be buffered, as follows:

- `_IOFBF` causes input/output to be fully buffered.
- `_IOLBF` causes output to be line buffered (the buffer will be flushed when a newline character is written, when the buffer is full, or when interactive input is requested).
- `_IONBF` causes input/output to be completely unbuffered.

If *buf* is not the null pointer, the array it points to may be used instead of an automatically allocated buffer (the buffer must have a lifetime at least as great as the open stream, so the stream should be closed before a buffer that has automatic storage duration is deallocated upon block exit). The argument *size* specifies the size of the array. The contents of the array at any time are indeterminate.

Returns: zero on success, or non-zero if an invalid value is given for *mode* or *size*, or if the request cannot be honoured.

Entry no. 92: int fprintf(FILE **stream*, const char **format*, ...)

Writes output to the stream pointed to by *stream*, under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behaviour is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The `fprintf` function returns when the end of the format string is reached. The format must be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not `%`), which are copied unchanged to the output stream; and conversion specifiers, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character `%`. For a complete description of the available conversion specifiers refer to section 4.9.6.1 in the ANSI standard. The minimum value for the maximum number of characters that can be produced by any single conversion is at least 509.

A brief and incomplete description of conversion specifications is:

`[flags][field width][.precision]specifier-char`

<i>flags</i>	is most commonly <code>-</code> , indicating left justification of the output item within the field. If omitted, the item will be right justified.
<i>field width</i>	is the minimum width of field to use. If the formatted item is longer, a bigger field will be used; otherwise, the item will be right (left) justified in the field.

precision is the minimum number of digits to print for a d, i, o, u, x or X conversion, the number of digits to appear after the decimal digit for e, E and f conversions, the maximum number of significant digits for g and G conversions, or the maximum number of characters to be written from strings in an s conversion.

Either of both of *field width* and *precision* may be *, indicating that the value is an argument to printf.

The *specifier chars* are:

d, i	int printed as signed decimal
o, u, x, X	unsigned int value printed as unsigned octal, decimal or hexadecimal
f	double value printed in the style [-]ddd.ddd
e, E	double value printed in the style [-]d.ddd...e±dd
g, G	double printed in f or e format, whichever is more appropriate
c	int value printed as unsigned char
s	char *value printed as a string of characters
p	void *argument printed as a hexadecimal address
%	write a literal %

Returns: the number of characters transmitted, or a negative value if an output error occurred.

Entry no. 91: int printf(const char *format, ...)

Equivalent to fprintf with the argument stdout interposed before the arguments to printf.

Returns: the number of characters transmitted, or a negative value if an output error occurred.

Entry no. 93: int sprintf(char *s, const char *format, ...)

Equivalent to fprintf, except that the argument *s* specifies an array into which the generated output is to be written, rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned sum.

Returns: the number of characters written to the array, not counting the terminating null character.

Entry no. 26: int _printf(const char *format, ...)

This function is identical in function to printf except that it does not handle floating point arguments.

It is used for space optimisation by the C compiler when using the non shared library and when a literal format string does not contain any floating point conversions.

It is included in the shared library for compatibility with the non shared library.

Entry no. 27: `int _fprintf(FILE *stream, const char *format, ...)`

This function is identical in function to `fprintf` except that it does not handle floating point arguments.

It is used for space optimisation by the C compiler when using the non shared library and when a literal format string does not contain any floating point conversions.

It is included in the shared library for compatibility with the non shared library.

Entry no. 28: `int _sprintf(char *s, const char *format, ...)`

This function is identical in function to `sprintf` except that it does not handle floating point arguments.

It is used for space optimisation by the C compiler when using the non shared library and when a literal format string does not contain any floating point conversions.

It is included in the shared library for compatibility with the non shared library.

Entry no. 100: `int _vfprintf(FILE *stream, const char *format, va_list arg)`

This function is identical in function to `vfprintf` except that it does not handle floating point arguments.

It is used for space optimisation by the C compiler when using the non shared library and when a literal format string does not contain any floating point conversions.

It is included in the shared library for compatibility with the non shared library.

Entry no. 95: `int fscanf(FILE *stream, const char *format, ...)`

Reads input from the stream pointed to by *stream*, under control of the string pointed to by *format* that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behaviour is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The format is composed of zero or more directives, one or more white-space characters, an ordinary character (not `%`), or a conversion specification. Each conversion specification is introduced by the character `%`. For a description of the available conversion specifiers refer to section 4.9.6.2 in the ANSI standard, or to any of the references listed in the chapter entitled *Introduction* on page 1 of the Acorn Desktop C Manual. A brief list is given above, under the entry for `fprintf`.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversions terminate on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including newline characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

Returns: the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `fscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early conflict between an input character and the format.

Entry no. 94: `int scanf(const char *format, ...)`

Equivalent to `fscanf` with the argument `stdin` interposed before the arguments to `scanf`.

Returns: the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `scanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Entry no. 96: `int sscanf(const char *s, const char *format, ...)`

Equivalent to `fscanf` except that the argument `s` specifies a string from which the input is to be obtained, rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the `fscanf` function.

Returns: the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `scanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Entry no. 97: `int vprintf(const char *format, va_list arg)`

Equivalent to `printf`, with the variable argument list replaced by `arg`, which has been initialised by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vprintf` function does not invoke the `va_end` function.

Returns: the number of characters transmitted, or a negative value if an output error occurred.

Entry no. 98: int vfprintf(FILE **stream*, const char **format*, va_list *arg*)

Equivalent to fprintf, with the variable argument list replaced by *arg*, which has been initialised by the va_start macro (and possibly subsequent va_arg calls). The vfprintf function does not invoke the va_end function.

Returns: the number of characters transmitted, or a negative value if an output error occurred.

Entry no. 99: int vsprintf(char **s*, const char **format*, va_list *arg*)

Equivalent to sprintf, with the variable argument list replaced by *arg*, which has been initialised by the va_start macro (and possibly subsequent va_arg calls). The vsprintf function does not invoke the va_end function.

Returns: the number of characters written in the array, not counting the terminating null character.

Entry no. 101: int fgetc(FILE **stream*)

Obtains the next character (if present) as an unsigned char converted to an int, from the input stream pointed to by *stream*, and advances the associated file position indicator (if defined).

Returns: the next character from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator is set and fgetc returns EOF. If a read error occurs, the error indicator is set and fgetc returns EOF.

Entry no. 102: char *fgets(char **s*, int *n*, FILE **stream*)

Reads at most one less than the number of characters specified by *n* from the stream pointed to by *stream* into the array pointed to by *s*. No additional characters are read after a newline character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

Returns: *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

Entry no. 103: int fputc(int *c*, FILE **stream*)

Writes the character specified by *c* (converted to an unsigned char) to the output stream pointed to by *stream*, at the position indicated by the associated file position indicator (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Returns: the character written. If a write error occurs, the error indicator is set and `fputc` returns EOF.

Entry no. 104: `int fputs(const char *s, FILE *stream)`

Writes the string pointed to by *s* to the stream pointed to by *stream*. The terminating null character is not written.

Returns: EOF if a write error occurs; otherwise it returns a non-negative value.

Entry no. 106: `int getc(FILE *stream)`

Equivalent to `fgetc` except that it may be (and is under RISC OS) implemented as a macro. *stream* may be evaluated more than once, so the argument should never be an expression with side effects.

Returns: the next character from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator is set and `getc` returns EOF. If a read error occurs, the error indicator is set and `getc` returns EOF.

Entry no. 107: `int getchar(void)`

Equivalent to `getc` with the argument `stdin`.

Returns: the next character from the input stream pointed to by `stdin`. If the stream is at end-of-file, the end-of-file indicator is set and `getchar` returns EOF. If a read error occurs, the error indicator is set and `getchar` returns EOF.

Entry no. 108: `char *gets(char *s)`

Reads characters from the input stream pointed to by `stdin` into the array pointed to by *s*, until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array.

Returns: *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

Entry no. 110: `int putc(int c, FILE *stream)`

Equivalent to `fputc` except that it may be (and is under RISC OS) implemented as a macro. *stream* may be evaluated more than once, so the argument should never be an expression with side effects.

Returns: the character written. If a write error occurs, the error indicator is set and `putc` returns EOF.

Entry no. 111: `int putchar(int c)`

Equivalent to `putc` with the second argument `stdout`.

Returns: the character written. If a write error occurs, the error indicator is set and `putc` returns EOF.

Entry no. 112: `int puts(const char *s)`

Writes the string pointed to by `s` to the stream pointed to by `stdout`, and appends a newline character to the output. The terminating null character is not written.

Returns: EOF if a write error occurs; otherwise it returns a non-negative value.

Entry no. 113: `int ungetc(int c, FILE *stream)`

Pushes the character specified by `c` (converted to an unsigned char) back onto the input stream pointed to by `stream`. The character will be returned by the next read on that stream. An intervening call to the `fflush` function or to a file positioning function (`fseek`, `fsetpos`, `rewind`) discards any pushed-back characters. The external storage corresponding to the stream is unchanged. One character pushback is guaranteed. If the `ungetc` function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail. If the value of `c` equals that of the macro `EOF`, the operation fails and the input stream is unchanged.

A successful call to the `ungetc` function clears the end-of-file indicator. The value of the file position indicator after reading or discarding all pushed-back characters will be the same as it was before the characters were pushed back. For a text stream, the value of the file position indicator after a successful call to the `ungetc` function is unspecified until all pushed-back characters are read or discarded. For a binary stream, the file position indicator is decremented by each successful call to the `ungetc` function; if its value was zero before a call, it is indeterminate after the call.

Returns: the character pushed back after conversion, or EOF if the operation fails.

Entry no. 114: `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)`

Reads into the array pointed to by `ptr`, up to `nmemb` members whose size is specified by `size`, from the stream pointed to by `stream`. The file position indicator (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting

value of the file position indicator is indeterminate. If a partial member is read, its value is indeterminate. The `ferror` or `feof` function shall be used to distinguish between a read error and end-of-file.

Returns: the number of members successfully read, which may be less than *nmemb* if a read error or end-of-file is encountered. If *size* or *nmemb* is zero, `fread` returns zero and the contents of the array and the state of the stream remain unchanged.

Entry no. 115: `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)`

Writes, from the array pointed to by *ptr* up to *nmemb* members whose size is specified by *size*, to the stream pointed to by *stream*. The file position indicator (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator is indeterminate.

Returns: the number of members successfully written, which will be less than *nmemb* only if a write error is encountered.

Entry no. 116: `int fgetpos(FILE *stream, fpos_t *pos)`

Stores the current value of the file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The value stored contains unspecified information usable by the `fsetpos` function for repositioning the stream to its position at the time of the call to the `fgetpos` function.

Returns: zero, if successful. Otherwise non-zero is returned and the integer expression `errno` is set to an implementation-defined non-zero value (under RISC OS `fgetpos` cannot fail).

Entry no. 117: `int fseek(FILE *stream, long int offset, int whence)`

Sets the file position indicator for the stream pointed to by *stream*. For a binary stream, the new position is at the signed number of characters specified by *offset* away from the point specified by *whence*. The specified point is the beginning of the file for `SEEK_SET`, the current position in the file for `SEEK_CUR`, or end-of-file for `SEEK_END`. A binary stream need not meaningfully support `fseek` calls with a *whence* value of `SEEK_END`, though the Acorn implementation does. For a text stream, *offset* is either zero or a value returned by an earlier call to the `ftell` function on the same stream; *whence* is then `SEEK_SET`. The Acorn implementation also allows a text stream to be positioned in exactly the same manner as a binary stream, but this is not portable. The `fseek` function clears the end-of-file indicator and undoes any effects of the `ungetc` function on the same stream. After an `fseek` call, the next operation on an update stream may be either input or output.

Returns: non-zero only for a request that cannot be satisfied.

Entry no. 118: int fsetpos(FILE **stream*, const fpos_t **pos*)

Sets the file position indicator for the stream pointed to by *stream* according to the value of the object pointed to by *pos*, which is a value returned by an earlier call to the `fgetpos` function on the same stream. The `fsetpos` function clears the end-of-file indicator and undoes any effects of the `ungetc` function on the same stream. After an `fsetpos` call, the next operation on an update stream may be either input or output.

Returns: zero, if successful. Otherwise non-zero is returned and the integer expression `errno` is set to an implementation-defined non-zero value (under RISC OS the value is that of `EDOM` in `math.h`).

Entry no. 119: long int ftell(FILE **stream*)

Obtains the current value of the file position indicator for the stream pointed to by *stream*. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, the file position indicator contains unspecified information, usable by the `fseek` function for returning the file position indicator to its position at the time of the `ftell` call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read. However, for the Acorn implementation, the value returned is merely the byte offset into the file, whether the stream is text or binary.

Returns: if successful, the current value of the file position indicator. On failure, the `ftell` function returns `-1L` and sets the integer expression `errno` to an implementation-defined non-zero value (under RISC OS `ftell` cannot fail).

Entry no. 120: void rewind(FILE **stream*)

Sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to `(void)fseek(stream, 0L, SEEK_SET)` except that the error indicator for the stream is also cleared.

Returns: no value.

Entry no. 121: void clearerr(FILE **stream*)

Clears the end-of-file and error indicators for the stream pointed to by *stream*. These indicators are cleared only when the file is opened or by an explicit call to the `clearerr` function or to the `rewind` function.

Returns: no value.

Entry no. 122: int feof(FILE **stream*)

Tests the end-of-file indicator for the stream pointed to by *stream*.

Returns: non-zero if the end-of-file indicator is set for *stream*.

Entry no. 123: int ferror(FILE *stream)

Tests the error indicator for the stream pointed to by *stream*.

Returns: non-zero if the error indicator is set for *stream*.

Entry no. 124: void perror(const char *s)

Maps the error number in the integer expression *errno* to an error message. It writes a sequence of characters to the standard error stream thus: first (if *s* is not a null pointer and the character pointed to by *s* is not the null character), the string pointed to by *s* followed by a colon and a space; then an appropriate error message string followed by a newline character. The contents of the error message strings are the same as those returned by the *strerror* function with argument *errno*, which are implementation-defined.

Returns: no value.

Entry no. 105: int __filbuf(FILE *stream)

This function is used by the C library to implement the 'getc' macro. The definition of the 'getc' macro is as follows:

```
#define getc(p) \
    (--(p)->_icnt) >= 0 ? *((p)->_ptr)++ : __filbuf(p)
```

where *p* is a pointer to a FILE structure.

__filbuf fills the buffer associated with *p* from a file stream and returns the first character of the buffer incrementing the buffer pointer and decrementing the input character count.

Entry no. 109: int __flsbuf(int ch, FILE *stream)

This function is used by the C library to implement the *putc* macro. The definition of the *putc* macro is as follows:

```
#define putc(ch, p) \
    (--(p)->_ocnt) >= 0 ? (*(p)->_ptr)++ = (ch) : __flsbuf(ch,p)
```

where *p* is a pointer to a FILE structure.

__flsbuf flushes the buffer associated with *p* to a file stream and writes the character *ch* to the file stream. The buffer pointer and output character count are reset.

stdlib

stdlib provides several general purpose functions

Entry no. 58: **double atof(const char *nptr)**

Converts the initial part of the string pointed to by *nptr* to double * representation.

Returns: the converted value.

Entry no. 59: **int atoi(const char *nptr)**

Converts the initial part of the string pointed to by *nptr* to int representation.

Returns: the converted value.

Entry no. 60: **long int atol(const char *nptr)**

Converts the initial part of the string pointed to by *nptr* to long int representation.

Returns: the converted value.

Entry no. 61: **double strtod(const char *nptr, char **endptr)**

Converts the initial part of the string pointed to by *nptr* to double representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function), a subject sequence resembling a floating point constant, and a final string of one or more unrecognised characters, including the terminating null character of the input string. It then attempts to convert the subject sequence to a floating point number, and returns the result. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Returns: the converted value if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned (according to the sign of the value), and the value of the macro `ERANGE` is stored in `errno`. If the correct value would cause underflow, zero is returned and the value of the macro `ERANGE` is stored in `errno`.

Entry no. 62: **long int strtol(const char *nptr, char **endptr, int base)**

Converts the initial part of the string pointed to by *nptr* to long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function), a subject sequence resembling an integer represented in some radix determined by the value of *base*, and a final string of one or more unrecognised characters, including the terminating null character of the input string.

It then attempts to convert the subject sequence to an integer, and returns the result. If the value of *base* is 0, the expected form of the subject sequence is that of an integer constant (described precisely in the ANSI standard, section 3.1.3.2), optionally preceded by a + or - sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the base are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits following the sign if present. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Returns: the converted value if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `LONG_MAX` or `LONG_MIN` is returned (according to the sign of the value), and the value of the macro `ERANGE` is stored in `errno`.

Entry no. 63: unsigned long int strtoul(const char *nptr, char **endptr, int base)

Converts the initial part of the string pointed to by *nptr* to unsigned long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white space characters (as determined by the `isspace` function), a subject sequence resembling an unsigned integer represented in some radix determined by the value of *base*, and a final string of one or more unrecognised characters, including the terminating null character of the input string.

It then attempts to convert the subject sequence to an unsigned integer, and returns the result. If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant (described precisely in the ANSI Standard, section 3.1.3.2), optionally preceded by a + or - sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a + or - sign, but not including an integer suffix. The letters from a (or A) through z (or Z) stand for the values 10 to 35; only letters whose ascribed values are less than that of the base are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits following the sign, if present. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Returns: the converted value if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `ULONG_MAX` is returned, and the value of the * macro `ERANGE` is stored in `errno`.

Entry no. 64: int rand(void)

Computes a sequence of pseudo-random integers in the range 0 to RAND_MAX, where RAND_MAX = 0x7fffffff.

Returns: a pseudo-random integer.

Entry no. 65: void srand(unsigned int seed)

Uses its argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand. If srand is then called with the same seed value, the sequence of pseudo-random numbers will be repeated. If rand is called before any calls to srand have been made, the same sequence is generated as when srand is first called with a seed value of 1.

Entry no. 66: void *calloc(size_t nmemb, size_t size)

Allocates space for an array of *nmemb* objects, each of whose size is *size*. The space is initialised to all bits zero.

Returns: either a null pointer or a pointer to the allocated space.

Entry no. 67: void free(void *ptr)

Causes the space pointed to by *ptr* to be deallocated (made available for further allocation). If *ptr* is a null pointer, no action occurs. Otherwise, if *ptr* does not match a pointer earlier returned by calloc, malloc or realloc or if the space has been deallocated by a call to free or realloc, the behaviour is undefined.

Entry no. 68: void *malloc(size_t size)

Allocates space for an object whose size is specified by *size* and whose value is indeterminate.

Returns: either a null pointer or a pointer to the allocated space.

Entry no. 69: void *realloc(void *ptr, size_t size)

Changes the size of the object pointed to by *ptr* to the size specified by *size*. The contents of the object is unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If *ptr* is a null pointer, the realloc function behaves like a call to malloc for the specified size. Otherwise, if *ptr* does not match a pointer earlier returned by calloc, malloc or realloc, or if the space has been deallocated by a call to free or realloc, the behaviour is undefined. If the space cannot be allocated, the object pointed to by *ptr* is unchanged. If size is zero and *ptr* is not a null pointer, the object it points to is freed.

Returns: either a null pointer or a pointer to the possibly moved allocated space.

Entry no. 70: void abort(void)

Causes abnormal program termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return. Whether open output streams are flushed or open streams are closed or temporary files removed is implementation-defined (under RISC OS all these occur). An implementation-defined form of the status ‘unsuccessful termination’ (1 under RISC OS) is returned to the host environment by means of a call to raise(SIGABRT).

Entry no. 71: int atexit(void (*func)(void))

Registers the function pointed to by *func*, to be called without its arguments at normal program termination. It is possible to register at least 32 functions.

Returns: zero if the registration succeeds, non-zero if it fails.

Entry no. 72: void exit(int status)

Causes normal program termination to occur. If more than one call to the exit function is executed by a program (for example, by a function registered with atexit), the behaviour is undefined. First, all functions registered by the atexit function are called, in the reverse order of their registration. Next, all open output streams are flushed, all open streams are closed, and all files created by the tmpfile function are removed. Finally, control is returned to the host environment. If the value of *status* is zero or EXIT_SUCCESS, an implementation-defined form of the status ‘successful termination’ (0 under RISC OS) is returned. If the value of *status* is EXIT_FAILURE, an implementation-defined form of the status ‘unsuccessful termination’ (1 under RISC OS) is returned. Otherwise the status returned is implementation-defined (the value of *status* is returned under RISC OS).

Entry no. 73: char *getenv(const char *name)

Searches the environment list, provided by the host environment, for a string that matches the string pointed to by *name*. The set of environment names and the method for altering the environment list are implementation-defined.

Returns: a pointer to a string associated with the matched list member. The array pointed to is not modified by the program, but may be overwritten by a subsequent call to the getenv function. If the specified name cannot be found, a null pointer is returned.

Entry no. 74: `int system(const char *string)`

Passes the string pointed to by *string* to the host environment to be executed by a command processor in an implementation-defined manner. A null pointer may be used for *string*, to inquire whether a command processor exists. Under RISC OS, care must be taken, when executing a command, that the command does not overwrite the calling program. To control this, the string chain: or call: may immediately precede the actual command. The effect of call: is the same as if call: were not present. When a command is called, the caller is first moved to a safe place in application workspace. When the callee terminates, the caller is restored. This requires enough memory to hold caller and callee simultaneously. When a command is chained, the caller may be overwritten. If the caller is not overwritten, the caller exits when the callee terminates. Thus a transfer of control is effected and memory requirements are minimised.

Returns: If the argument is a null pointer, the system function returns non-zero only if a command processor is available. If the argument is not a null pointer, it returns an implementation-defined value (under RISC OS 0 is returned for success and -2 for failure to invoke the command; any other value is the return code from the executed command).

Entry no. 75: `void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))`

Searches an array of *nmemb* objects, the initial member of which is pointed to by *base*, for a member that matches the object pointed to by *key*. The size of each member of the array is specified by *size*. The contents of the array must be in ascending sorted order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the key object and to an array member, in that order. The function returns an integer less than, equal to, or greater than zero if the key object is considered, respectively, to be less than, to match, or to be greater than the array member.

Returns: a pointer to a matching member of the array, or a null pointer if no match is found. If two members compare as equal, which member is matched is unspecified.

Entry no. 76: `void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))`

Sorts an array of *nmemb* objects, the initial member of which is pointed to by *base*. The size of each object is specified by *size*. The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared. The function returns an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is unspecified.

Entry no. 77: int abs(int j)

Computes the absolute value of an integer *j*. If the result cannot be represented, the behaviour is undefined.

Returns: the absolute value.

Entry no. 78: div_t div(int numer, int denom)

Computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behaviour is undefined; otherwise, $\text{quot} * \text{denom} + \text{rem} = \text{numer}$.

Returns: a structure of type `div_t`, comprising both the quotient and the remainder. The structure contains the following members: `int quot`; `int rem`. You may not rely on their order.

Entry no. 79: long int labs(long int j)

Computes the absolute value of a long integer *j*. If the result cannot be represented, the behaviour is undefined.

Returns: the absolute value.

Entry no. 80: ldiv_t ldiv(long int numer, long int denom)

Computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented, the behaviour is undefined; otherwise, $\text{quot} * \text{denom} + \text{rem} = \text{numer}$.

Returns: a structure of type `ldiv_t`, comprising both the quotient and the remainder. The structure contains the following members: `long int quot`; `long int rem`. You may not rely on their order.

Multibyte character functions

The behaviour of the multibyte character functions is affected by the `LC_CTYPE` category of the current locale. For a state-dependent encoding, each function is placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer causes these functions to return a non-zero value if encoding have state dependency, and a zero otherwise. After the `LC_CTYPE` category is changed, the shift state of these functions is indeterminate.

Entry no. 172: int mblen(const char *s, size_t n)

If *s* is not a null pointer, the `mblen` function determines the number of bytes comprising the multibyte character pointed to by *s*. Except that the shift state of the `mbtowc` function is not affected, it is equivalent to `mbtowc((wchar_t *)0, s, n)`.

Returns: If *s* is a null pointer, the `mblen` function returns a non-zero or zero value, if multibyte character encodings, respectively do or do not have state-dependent encodings. If *s* is not a null pointer, the `mblen` function either returns a 0 (if *s* points to a null character), or returns the number of bytes that comprise the multibyte character (if the next *n* or fewer bytes form a valid multibyte character), or returns `-1` (if they do not form a valid multibyte character).

Entry no. 173: int mbtowc(wchar_t *pwc, const char *s, size_t n)

If *s* is not a null pointer, the `mbtowc` function determines the number of bytes that comprise the multibyte character pointed to by *s*. It then determines the code for value of type `wchar_t` that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero). If the multibyte character is valid and *pwc* is not a null pointer, the `mbtowc` function stores the code in the object pointed to by *pwc*. At most *n* bytes of the array pointed to by *s* will be examined.

Returns: If *s* is a null pointer, the `mbtowc` function returns a non-zero or zero value, if multibyte character encodings, respectively do or do not have state-dependent encodings. If *s* is not a null pointer, the `mbtowc` function either returns a 0 (if *s* points to a null character), or returns the number of bytes that comprise the converted multibyte character (if the next *n* or fewer bytes form a valid multibyte character), or returns `-1` (if they do not form a valid multibyte character).

Entry no. 174: int wctomb(char *s, wchar_t wchar)

Determines the number of bytes need to represent the multibyte character corresponding to the code whose value is *wchar* (including any change in shift state). It stores the multibyte character representation in the array object pointed to by *s* (if *s* is not a null pointer). At most `MB_CUR_MAX` characters are stored. If the value of *wchar* is zero, the `wctomb` function is left in the initial shift state).

Returns: If *s* is a null pointer, the `wctomb` function returns a non-zero or zero value, if multibyte character encodings, respectively do or do not have state-dependent encodings. If *s* is not a null pointer, the `wctomb` function returns a `-1` if the value of *wchar* does not correspond to a valid multibyte character, or returns the number of bytes that comprise the multibyte character corresponding to the value of *wchar*.

Multibyte string functions

The behaviour of the multibyte string functions is affected by the LC_CTYPE category of the current locale.

Entry no. 175: `size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n)`

Converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by *s* into a sequence of corresponding codes and stores not more than *n* codes into the array pointed to by *pwcs*. No multibyte character that follows a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to the `mbtowc` function. If an invalid multibyte character is found, `mbstowcs` returns `(size_t)-1`. Otherwise, the `mbstowcs` function returns the number of array elements modified, not including a terminating zero code, if any.

Entry no. 176: `size_t wcstombs(char *s, const wchar_t *pwcs, size_t n)`

Converts a sequence of codes that correspond to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array pointed to by *s*, stopping if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to the `wctomb` function, except that the shift state of the `wctomb` function is not affected. If a code is encountered which does not correspond to any valid multibyte character, the `wcstombs` function returns `(size_t)-1`. Otherwise, the `wcstombs` function returns the number of bytes modified, not including a terminating null character, if any.

string

string provides several functions useful for manipulating character arrays and other objects treated as character arrays. Various methods are used for determining the lengths of the arrays, but in all cases a `char *` or `void *` argument points to the initial (lowest addresses) character of the array. If an array is written beyond the end of an object, the behaviour is undefined.

Entry no. 38: `void *memcpy(void *s1, const void *s2, size_t n)`

Copies n characters from the object pointed to by $s2$ into the object pointed to by $s1$. If copying takes place between objects that overlap, the behaviour is undefined.

Returns: the value of $s1$.

Entry no. 39: `void *memmove(void *s1, const void *s2, size_t n)`

Copies n characters from the object pointed to by $s2$ into the object pointed to by $s1$. Copying takes place as if the n characters from the object pointed to by $s2$ are first copied into a temporary array of n characters that does not overlap the objects pointed to by $s1$ and $s2$, and then the n characters from the temporary array are copied into the object pointed to by $s1$.

Returns: the value of $s1$.

Entry no. 40: `char *strcpy(char *s1, const char *s2)`

Copies the string pointed to by $s2$ (including the terminating null character) into the array pointed to by $s1$. If copying takes place between objects that overlap, the behaviour is undefined.

Returns: the value of $s1$.

Entry no. 41: `char *strncpy(char *s1, const char *s2, size_t n)`

Copies not more than n characters (characters that follow a null character are not copied) from the array pointed to by $s2$ into the array pointed to by $s1$. If copying takes place between objects that overlap, the behaviour is undefined. If terminating nul has not been copied in chars, no term nul is placed in $s2$.

Returns: the value of $s1$.

Entry no. 42: char *strcat(char *s1, const char *s2)

Appends a copy of the string pointed to by *s2* (including the terminating null character) to the end of the string pointed to by *s1*. The initial character of *s2* overwrites the null character at the end of *s1*.

Returns: the value of *s1*.

Entry no. 43: char *strncat(char *s1, const char *s2, size_t n)

Appends not more than *n* characters (a null character and characters that follow it are not appended) from the array pointed to by *s2* to the end of the string pointed to by *s1*. The initial character of *s2* overwrites the null character at the end of *s1*. A terminating null character is always appended to the result.

Returns: the value of *s1*.

The sign of a non-zero value returned by the comparison functions is determined by the sign of the difference between the values of the first pair of characters (both interpreted as unsigned char) that differ in the objects being compared.

Entry no. 44: int memcmp(const void *s1, const void *s2, size_t n)

Compares the first *n* characters of the object pointed to by *s1* to the first *n* characters of the object pointed to by *s2*.

Returns: an integer greater than, equal to, or less than zero, depending on whether the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*.

Entry no. 45: int strcmp(const char *s1, const char *s2)

Compares the string pointed to by *s1* to the string pointed to by *s2*.

Returns: an integer greater than, equal to, or less than zero, depending on whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*.

Entry no. 46: int strncmp(const char *s1, const char *s2, size_t n)

Compares not more than *n* characters (characters that follow a null character are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.

Returns: an integer greater than, equal to, or less than zero, depending on whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*.

Entry no. 178: int strcoll(const char *s1, const char *s2)

Compares the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the LC_COLLATE category of the current locale.

Returns: an integer greater than, equal to, or less than zero, depending on whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the current locale.

Entry no. 177: `size_t strxfrm(char *s1, const char *s2, size_t n)`

Transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation function is such that if the `strcmp` function is applied to two transformed strings, it returns a value greater than, equal to or less than zero, corresponding to the result of the `strcoll` function applied to the same two original strings. No more than *n* characters are placed into the resulting array pointed to by *s1*, including the terminating null character. If *n* is zero, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behaviour is undefined.

Under RISC OS 3 (version 3.10) this function only works for the default ANSI locale, but not for other locales (ie not after a `setlocale` call).

Returns: The length of the transformed string is returned (not including the terminating null character). If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate.

Entry no. 47: `void *memchr(const void *s, int c, size_t n)`

Locates the first occurrence of *c* (converted to an unsigned char) in the initial *n* characters (each interpreted as unsigned char) of the object pointed to by *s*.

Returns: a pointer to the located character, or a null pointer if the character does not occur in the object.

Entry no. 48: `char *strchr(const char *s, int c)`

Locates the first occurrence of *c* (converted to a char) in the string pointed to by *s* (including the terminating null character). The BSD UNIX name for this function is `index()`.

Returns: a pointer to the located character, or a null pointer if the character does not occur in the string.

Entry no. 49: `size_t strcspn(const char *s1, const char *s2)`

Computes the length of the initial segment of the string pointed to by *s1* which consists entirely of characters not from the string pointed to by *s2*. The terminating null character is not considered part of *s2*.

Returns: the length of the segment.

Entry no. 50: `char *strpbrk(const char *s1, const char *s2)`

Locates the first occurrence in the string pointed to by *s1* of any character from the string pointed to by *s2*.

Returns: returns a pointer to the character, or a null pointer if no character from *s2* occurs in *s1*.

Entry no. 51: `char *strrchr(const char *s, int c)`

Locates the last occurrence of *c* (converted to a char) in the string pointed to by *s*. The terminating null character is considered part of the string. The BSD UNIX name for this function is `rindex()`.

Returns: a pointer to the character, or a null pointer if *c* does not occur in the string.

Entry no. 52: `size_t strspn(const char *s1, const char *s2)`

Computes the length of the initial segment of the string pointed to by *s1* which consists entirely of characters from the string pointed to by *s2*.

Returns: the length of the segment.

Entry no. 53: `char *strstr(const char *s1, const char *s2)`

Locates the first occurrence in the string pointed to by *s1* of the sequence of characters (excluding the terminating null character) in the string pointed to by *s2*.

Returns: a pointer to the located string, or a null pointer if the string is not found.

Entry no. 54: `char *strtok(char *s1, const char *s2)`

A sequence of calls to the `strtok` function breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *s2*. The first call in the sequence has *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *s2* may be different from call to call. The first call in the sequence searches for the first character that is not contained in the current separator string *s2*. If no such character is found, then there are no tokens in *s1* and the `strtok` function returns a null pointer. If such a character is found, it is the start of the first token. The `strtok` function then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searches for a token will fail. If such a character is found, it is overwritten by a null character, which terminates the current token. The `strtok` function saves a pointer to the following character, from which the next search for a token will start. Each subsequent call, with a null pointer as the value for the first argument, starts searching from the saved pointer and behaves as described above.

Returns: pointer to the first character of a token, or a null pointer if there is no token.

Entry no. 55: `void *memset(void *s, int c, size_t n)`

Copies the value of *c* (converted to an unsigned char) into each of the first *n* characters of the object pointed to by *s*.

Returns: the value of *s*.

Entry no. 56: `char *strerror(int errnum)`

Maps the error number in *errnum* to an error message string.

Returns: a pointer to the string, the contents of which are implementation-defined.

Under RISC OS and Arthur the strings for the given *errnums* are as follows:

- 0 No error (errno = 0)
- EDOM EDOM – function argument out of range
- ERANGE ERANGE – function result not representable
- ESIGNUM ESIGNUM – illegal signal number to signal() or raise()
- others Error code (errno) has no associated message.

The array pointed to may not be modified by the program, but may be overwritten by a subsequent call to the `strerror` function.

Entry no. 57: `size_t strlen(const char *s)`

Computes the length of the string pointed to by *s*.

Returns: the number of characters that precede the terminating null character.

time

`time` provides several functions for manipulating time. Many functions deal with a calendar time that represents the current date (according to the Gregorian calendar) and time. Some functions deal with local time, which is the calendar time expressed for some specific time zone, and with Daylight Saving Time, which is a temporary change in the algorithm for determining local time.

`struct tm` holds the components of a calendar time called the broken-down time. The value of `tm_isdst` is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

```
struct tm {
    int tm_sec; /* seconds after the minute, 0 to 60
                (0-60 allows for the occasional leap
                second) */
    int tm_min /* minutes after the hour, 0 to 59 */
    int tm_hour /* hours since midnight, 0 to 23 */
    int tm_mday /* day of the month, 0 to 31 */
    int tm_mon /* months since January, 0 to 11 */
    int tm_year /* years since 1900 */
    int tm_wday /* days since Sunday, 0 to 6 */
    int tm_yday /* days since January 1, 0 to 365 */
    int tm_isdst /* Daylight Saving Time flag */
};
```

Entry no. 29: clock_t clock(void)

Determines the processor time used.

Returns: the implementation's best approximation to the processor time used by the program since program invocation. The time in seconds is the value returned, divided by the value of the macro `CLOCKS_PER_SEC`. The value `(clock_t)-1` is returned if the processor time used is not available. In the desktop, `clock()` returns all processor time, not just that of the program.

Entry no. 30: double difftime(time_t time1, time_t time0)

Computes the difference between two calendar times: *time1* - *time0*. Returns: the difference expressed in seconds as a double.

Entry no. 31: time_t mktime(struct tm *timeptr)

Converts the broken-down time, expressed as local time, in the structure pointed to by `timeptr` into a calendar time value with the same encoding as that of the values returned by the `time` function. The original values of the `tm_wday` and `tm_yday` components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above. On successful completion, the values of the `tm_wday` and `tm_yday` structure components are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined.

Returns: the specified calendar time encoded as a value of type `time_t`. If the calendar time cannot be represented, the function returns the value `(time_t)-1`.

Entry no. 32: `time_t time(time_t *timer)`

Determines the current calendar time. The encoding of the value is unspecified.

Returns: the implementation's best approximation to the current calendar time. The value `(time_t)-1` is returned if the calendar time is not available. If *timer* is not a null pointer, the return value is also assigned to the object it points to.

Entry no. 33: `char *asctime(const struct tm *timeptr)`

Converts the broken-down time in the structure pointed to by *timeptr* into a string in the style `Sun Sep 16 01:03:52 1973\n\0`.

Returns: a pointer to the string containing the date and time.

Entry no. 34: `char *ctime(const time_t *timer)`

Converts the calendar time pointed to by *timer* to local time in the form of a string. It is equivalent to `asctime(localtime(timer))`.

Returns: the pointer returned by the `asctime` function with that broken-down time as argument.

Entry no. 35: `struct tm *gmtime(const time_t *timer)`

Converts the calendar time pointed to by *timer* into a broken-down time, expressed as Greenwich Mean Time (GMT).

Returns: a pointer to that object or a null pointer if GMT is not available.

Entry no. 36: `struct tm *localtime(const time_t *timer)`

Converts the calendar time pointed to by *timer* into a broken-down time, expressed a local time.

Returns: a pointer to that object.

Entry no. 37: `size_t strftime(char *s, size_t maxsize, const char *format, const struct tm *timeptr)`

Places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. The format string consists of zero or more directives and ordinary characters. A directive consists of a `%` character followed by a character that determines the directive's behaviour. All ordinary characters (including the terminating null character) are copied unchanged into the array. No more than *maxsize* characters are placed into

the array. Each directive is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the LC_TIME category of the current locale and by the values contained in the structure pointed to by `timeptr`.

Directive	Replaced by
<code>%a</code>	the locale's abbreviated weekday name
<code>%A</code>	the locale's full weekday name
<code>%b</code>	the locale's abbreviated month name
<code>%B</code>	the locale's full month name
<code>%c</code>	the locale's appropriate date and time representation
<code>%d</code>	the day of the month as a decimal number (01 - 31)
<code>%H</code>	the hour (24-hour clock) as a decimal number (00 - 23)
<code>%I</code>	the hour (12-hour clock) as a decimal number (01 - 12)
<code>%j</code>	the day of the year as a decimal number (001 - 366)
<code>%m</code>	the month as a decimal number (01 - 12)
<code>%M</code>	the minute as a decimal number (00 - 61)
<code>%p</code>	the locale's equivalent of either AM or PM designation associated with a 12-hour clock
<code>%S</code>	the second as a decimal number (00 - 61)
<code>%U</code>	the week number of the year (Sunday as the first day of week 1) as a decimal number (00 - 53)
<code>%w</code>	the weekday as a decimal number (0 (Sunday) - 6)
<code>%W</code>	the week number of the year (Monday as the first day of week 1) as a decimal number (00 - 53)
<code>%x</code>	the locale's appropriate date representation
<code>%X</code>	the locale's appropriate time representation
<code>%y</code>	the year without century as a decimal number (00 - 99)
<code>%Y</code>	the year with century as a decimal number
<code>%Z</code>	the time zone name or abbreviation, or by no character if no time zone is determinable
<code>%%</code>	<code>%</code>

If a directive is not one of the above, the behaviour is undefined.

Returns: If the total number of resulting characters including the terminating null character is not more than *maxsize*, the `strptime` function returns the number of characters placed into the array pointed to by *s* not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

Introduction and Overview

Facilities were added to BASIC (and to BASIC64) in RISC OS 3 so that its messages can be translated for use in another territory. The BASIC interpreter issues calls to the BASICTrans module, which is responsible for providing messages appropriate to a particular territory. By replacing one BASICTrans module with another, you can change the language used by BASIC for its messages.

Both BASIC and BASIC64 issue the same calls to the same BASICTrans module, thus code and messages are shared between the two modules.

If you write a BASICTrans module, you can allocate memory for the translation from the RMA:

- Memory inside the SWI call is invulnerable to the task swapping problem found when BASIC itself attempts to use RMA memory. ‘Task manager’ swapping between two BASIC programs does not occur when in SWI mode.

Using BBC BASIC

For the sake of completeness, this chapter documents the *BASIC and *BASIC64 commands used to enter BBC BASIC. For full details of using BBC BASIC, see the *BBC BASIC Reference Manual*, available from your Acorn supplier.

SWI Calls

BASICTrans_HELP (SWI &42C80)

Interpret, translate if required, and print HELP messages

On entry

R0 = pointer to lexically analysed HELP text (terminated by &0D)

R1 = pointer to program's name (BASIC or BASIC64)

R2 = pointer to the lexical analyser's tables

On exit

R0 - R2 corrupted

Use

This call is made by BASIC to request that a BASICTrans module print a help message. BASIC lexically analyses the HELP text, converting keywords to tokens, before making this call. The currently loaded BASICTrans module then prints appropriate help text.

On entry R1 points to the program's name, and so is non-zero; if it is still non-zero on exit BASIC will print its own (short, English) Help text. Consequently, a BASICTrans module will normally set R1 to zero on exit – but the English version of BASICTrans sometimes preserves R1 so that its own help is followed by the default help.

In order to share the entirety of the HELP text between BASIC and BASIC64, this call is implemented for English, and both BASIC and BASIC64 are assembled without their own HELP text. About 15Kbytes are shared like this.

BASICTrans_Error (SWI &42C81)

Copy translated error string to buffer

On entry

R0 = unique error number (0 - 112)

R1 = pointer to buffer in which to place the error

On exit

R0 - R3 corrupted

Use

This call is made by BASIC to request that a BASICTrans module provide an error message. The currently loaded BASICTrans module places a null terminated error string for the given error number in the buffer pointed to by R1. The error string is null terminated. BASIC then prints the error message, and performs other actions necessary to smoothly integrate the error message with BASIC's normal provisions for error handling.

An error is generated if the BASICTrans module is not present (ie the SWI is not found), or if BASICTrans does not perform the translation. BASIC then prints a default (English) message explaining this.

In order to share the entirety of the error string text between BASIC and BASIC64, this call is implemented for English, and both BASIC and BASIC64 are assembled without their error messages. About 6Kbytes are shared like this. Correct error numbers are vital to the functioning of the interpreter, and so – rather than being shared – these are held in BASIC or BASIC64.

BASICTrans_Message (SWI &42C82)

Translate and print miscellaneous message

On entry

R0 = unique message number (0 - 25)

R1 - R3 = message dependent values

On exit

R0, R1 corrupted

Use

This call is made by BASIC to request that the BASICTrans module print a 'miscellaneous' message. Further parameters are passed that depend on the message you require to be printed.

An error is generated if the BASICTrans module is not present (ie the SWI is not found), or if BASICTrans does not perform the translation. BASIC then prints the full (English) version of the message that it holds internally.

The English BASICTrans module behaves as if this call does not exist, so that the default messages get printed. There are not many 'miscellaneous' messages, so no great saving is to be had in providing RISC OS 3 with a shared implementation.

The classic problem of the error handler's ' at line ' can now be handled as follows:

```
TRACE OFF
IF QUIT=TRUE THEN
  ERROR EXT,ERR,REPORT$
ELSE
  RESTORE:!(HIMEM-4)=@%
  SYS"BASICTrans_Message",21,ERL,REPORT$ TO :@%
  IF (@% AND 1)<>0 THEN
    REPORT:@%=&900:IF ERL<>0 THEN PRINT" at line "ERL ELSE PRINT
  ENDIF
  @%=! (HIMEM-4)
ENDIF
END
```

This allows the BASICTrans_Message code to print the string and optional ' at line ' ERL information in any order it likes.

*BASIC -help

This command causes BASIC to print some help information describing the options documented here. Then BASIC starts as usual.

*BASIC [-chain] *filename*

If you give a *filename* after the *BASIC command, optionally preceded by the keyword -chain, then the named file is loaded and executed. When the program stops, BASIC enters immediate mode, as usual.

*BASIC -quit *filename*

This behaves in a similar way to the previous option. However, when the program terminates, BASIC quits automatically, returning to the environment from which the interpreter was originally called. It also performs a CRUNCH %1111 on the program (for further details see the description of the CRUNCH command in the *BBC BASIC Reference Manual*). This is the default action used by BASIC programs that are executed as * commands. In addition, the function QUIT returns TRUE if BASIC is called in this fashion.

*BASIC -load *filename*

This option causes the file to be loaded automatically, but not executed. BASIC remains in immediate mode, from where the program can be edited or executed as required.

*BASIC @*start,end*

This acts in a similar way to the -load form of the command. However, the program that is 'loaded' automatically is not in a file, but already in memory. Following the @ are two addresses. These give, in hexadecimal, the address of the start of the in-core program, and the address of the byte after the last one. The program is copied to PAGE and tokenised if necessary. This form of the command is used by Twin when returning to BASIC.

Note that the in-core address description is fixed format. It should be in the form:

@xxxxxxxx,xxxxxxxx

where *x* means a hexadecimal digit. Leading zeros must be supplied. The command line terminator character must come immediately after the last digit. No spaces are allowed.

*BASIC -chain @*start,end*

This behaves like the previous option, but the program is executed as well. When the program terminates, BASIC enters immediate mode.

*BASIC -quit @*start,end*

This option behaves as the previous one, but when the BASIC program terminates, BASIC automatically quits. The function QUIT will return TRUE during the execution of the program.

Examples

```
*BASIC
*BASIC -quit shellProg
*BASIC @000ADF0C,000AE345
*BASIC -chain fred
```

Related commands

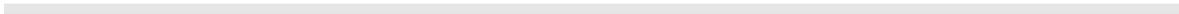
None

Related SWIs

None

Related vectors

None



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. With Edit you can set the type of the file to Command or Obey, except under RISC OS 2, where you then have to use the command `*SetType` .

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* on page 2-380).

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 on page 2-167.

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
```

Note that Obey files can also be nested, calling 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.

Making a script run automatically

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 RISC OS 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 section entitled *Application resource files* on page 3-56.
- From an application directory when the application is run.
The file must be called !Run. For further details see the section entitled *Application resource files* on page 3-56.

In the first two cases you will need to use the *Opt command as well (see page 2-179).

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 `*Set` on page 1-333.

Abbreviations

You must not use abbreviations for `*` Commands in scripts and programs, as these may vary between releases of RISC OS. For example, in RISC OS 2 '`*Te.`' was the minimum abbreviation for `*Tempo`, whereas in RISC OS 3 this abbreviation instead runs the `*Territories` command.

*Commands

*Obey

Executes a file of * commands

Syntax

*Obey [[-v][-c] [*filename* [*parameters*]]]

Parameters

-v	echo each line before execution
-c	cache <i>filename</i> , and execute it from memory
<i>filename</i>	a valid pathname, specifying a file
<i>parameters</i>	strings separated by spaces

Use

*Obey executes a file of * commands. Argument substitution is performed on each line, using parameters passed in the command.

With the -v option, each line is displayed before execution. With the -c option, the file is cached and executed from memory. These options are not available in RISC OS 2.

Example

```
*Obey !commands myfile1 12
```

Related commands

*Exec, *Run

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 in both Command and Obey files:

```
echo <7>
echo |<7>
```

The control characters are represented in GStrans format (see the chapter entitled *Conversions* on page 1-455). 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 *VDU 7* on page 1-577 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 VDU drivers – displays <7> on 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 page 1-584 onwards of 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.

Appendixes and tables



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 their 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.

Available assemblers

The BASIC assembler

The BBC BASIC interpreter, supplied as a standard part of RISC OS, includes an ARM assembler. This supports the full instruction set of the ARM 2 processor. At present it neither supports extra instructions that were first implemented by the ARM 3 processor, nor does it support coprocessor instructions.

It is the BASIC assembler that is described below, serving as an introduction to ARM assembler.

The Acorn Desktop Assembler

The Acorn Desktop Assembler is a separate product that provides much more powerful facilities than the BASIC assembler. With it you can develop assembler programs under the desktop, in an environment common to all Acorn desktop languages. It contains two different assemblers:

- **AAsm** is an assembler that produces binary image files which can be executed immediately.
- **ObjAsm** is an assembler that creates object files that cannot be executed directly, but must first be linked to other object files. Object files linked with those produced by ObjAsm may be produced from some programming language other than assembler, for example C.

These assemblers are not described in this appendix, but use a broadly similar syntax to the BASIC assembler described below. For full details, see the *Acorn Assembler Release 2* manual, which is supplied with Acorn Desktop Assembler, or is separately available.

The BASIC assembler

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:

```
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 to hold your assembled machine code 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, this example reserves a total of 101 bytes of memory. In future examples, the size of memory reserved is shown as *required_size*, to emphasise that you must substitute a value appropriate to the size of your code.

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% required_size
...
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 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.

Implementing passes

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% required_size
FOR pass% = 0 TO 3 STEP 3
    P% = code%
    [
        OPT pass%
        ...           further assembly language statements and assembler directives
    ]
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 not 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 has been reserved (ie none of it is held in an address greater than the value held in L%). When reserving space, L% might be set as follows:

DIM code% *required_size*
L% = code% + *required_size*

Saving machine code to file

Once an assembly language routine has been successfully assembled, you can then save it to file. To do so, you can use the *Save command. In our above examples, code% points to the start of the code; after assembly, P% points to the byte after the code. So we could use this BASIC command:

```
OSCLI "Save "+outfile$+" "+STR$~(code%)+ " "+STR$~(P%)
```

after the above example to save the code in the file named by outfile\$.

Executing a machine code program

From memory

From memory, 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.

From file

The commands below will load and run the named file, using either its filetype (such as &FF8 for absolute code) and the associated Alias\$@LoadType_xxx and Alias\$@RunType_xxx 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 <i>int</i>	Define 1 byte of memory from LSB of <i>int</i> (DCB, =)
EQUW <i>int</i>	Define 2 bytes of memory from <i>int</i> (DCW)
EQU4 <i>int</i>	Define 4 bytes of memory from <i>int</i> (DCD)
EQU5 <i>str</i>	Define 0 - 255 bytes as required by string expression <i>str</i> (DCS)
ALIGN	Align P% (and O%) to the next word (4 byte) boundary
ADR <i>reg,addr</i>	Assemble instruction to load <i>addr</i> into <i>reg</i>

- The first four operations initialise the reserved memory to the values specified by the operand. In the case of EQU\$ the operand field must be a string expression. In all other cases it must be a numeric expression. DCB (and =), 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 EQU\$ to ensure that the next instruction is word-aligned.
- ADR assembles a single instruction – typically but not necessarily an ADD or SUB – with reg as the destination register. It obtains addr in that register. It does so in a PC-relative (ie position independent) manner where possible.

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* on page 1-9.

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

You should not use the NV (never) condition code – see page 4-388.

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 that either x or y but not both may be given
#exp	indicates that a BASIC 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 (because the ARM does not handle overflow for signed arithmetic shifts), and synonyms. LSL is the preferred form, as it indicates the functionality.

Move instructions

Syntax:

opcode«cond»«S» Rd, (#explRm)«,shift»

There are two move instructions. ‘Op2’ means ‘(#explRm)«,shift»’:

Instruction		Calculation performed
MOV	Move	Rd = Op2
MOVN	Move NOT	Rd = NOT Op2

Each of these instructions produces a result which it places in a destination register (Rd). The instructions do not affect bytes in memory directly.

Again, 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. These instructions set N and Z from the ALU, C from the shifter (but only if it is used), and do not affect V.

Examples:

```
MOV R0, #10 ; Load R0 with the value 10.
```

Special actions are taken if the source register is R15; the action is as follows:

- If Rm=R15 all 32 bits of R15 are used in the operation ie the PC + PSR.

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.)

Arithmetic and logical instructions

Syntax:

opcode«cond»«S» Rd, Rn, (#explRm)««,shift»

The instructions available are given below; again, ‘Op2’ means ‘(#explRm)««,shift»’:

Instruction		Calculation performed
ADC	Add with carry	$Rd = Rn + Op2 + C$
ADD	Add without carry	$Rd = Rn + Op2$
SBC	Subtract with carry	$Rd = Rn - Op2 - (1 - C)$
SUB	Subtract without carry	$Rd = Rn - Op2$
RSC	Reverse subtract with carry	$Rd = Op2 - Rn - (1 - C)$
RSB	Reverse subtract without carry	$Rd = Op2 - Rn$
AND	Bitwise AND	$Rd = Rn \text{ AND } Op2$
BIC	Bitwise AND NOT	$Rd = Rn \text{ AND NOT } (Op2)$
ORR	Bitwise OR	$Rd = Rn \text{ OR } Op2$
EOR	Bitwise EOR	$Rd = Rn \text{ EOR } Op2$

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.
```

Special actions are taken if any of the source registers are R15; the action is as follows:

- If Rm=R15 all 32 bits of R15 are used in the operation ie the PC + PSR.
- If Rn=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.)

Comparison instructions

Syntax:

opcode«cond»«SIP» Rn, (#explRm)««,shift»

There are four comparison instructions; again, ‘Op2’ means ‘(#explRm)««,shift»’:

Instruction		Calculation performed
CMN	Compare negated	Rn + Op2
CMP	Compare	Rn – Op2
TEQ	Test equal	Rn EOR Op2
TST	Test	Rn AND Op2

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 opcode, 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, #&80000000    ; 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 ptr 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,
LDMFD  sp!, {r0, r1, pc}     ; restore 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	$Rd = Rm \times Rs$
MLA	Multiply-accumulate	$Rd = Rm \times Rs + Rn$

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.

Examples:

```

MUL     R1,R2,R3
MLAEQS R1,R2,R3,R4
    
```

Branching instructions

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 15 usable condition codes:

Instruction

B	Branch
BL	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
        .....

MOV5 R15,R14 ; end of subroutine
```

Single register load/save instructions

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, are 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.

Multiple register load/save instructions

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 or saved. The order the registers are given, in the list, is irrelevant since the lowest numbered register is loaded/saved first, and the highest numbered one last. For example, a list comprising {R5,R3,R1,R8} is loaded/saved 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 A fter each store/load
DB	Decrement Rn B efore each store/load
IA	Increment Rn A fter each store/load
IB	Increment Rn B efore each store/load
EA	Empty A scending stack is used
ED	Empty D escending stack is used
FA	Full A scending stack is used
FD	Full D escending 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	LMDA
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 use a full descending stack as well – 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 &1478
                                ;       R1 from &147C
                                ;       R2 from &1480

```

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:

```

STMFD Stackpointer!, {R0,R1,R2,R3} ; push onto stack
...
LDMFD 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.

However, you should see *Appendix B: Warnings on the use of ARM assembler* on page 4-383 for notes on using writeback when doing so.

- You can store 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.

SWI instruction**Syntax:**

```
SWI<cond> expression
```

```
SWI<cond> "SWIname"                (BBC BASIC assembler)
```

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 *An introduction to SWIs* on page 1-23 for a full description of how RISC OS handles SWIs, and the index of SWIs for a full list of the operating system SWIs.

Introduction

The ARM processor family uses Reduced Instruction Set (RISC) techniques to maximise performance; as such, the instruction set allows some instructions and code sequences to be constructed that will give rise to unexpected (and potentially erroneous) results. These cases must be avoided by all machine code writers and generators if correct program operation across the whole range of ARM processors is to be obtained.

In order to be upwards compatible with future versions of the ARM processor family **never** use any of the undefined instruction formats:

- those shown in the *Acorn RISC Machine family Data Manual* as 'Undefined' which the processor traps;
- those which are not shown in the manual and which don't trap (for example, a Multiply instruction where bit 5 or 6 of the instruction is set).

In addition the 'NV' (never executed) instruction class should not be used (it is recommended that the instruction 'MOV R0,R0' be used as a general purpose no-op).

This chapter lists the instructions and code sequences to be avoided. It is **strongly** recommended that you take the time to familiarise yourself with these cases because some will only fail under particular circumstances which may not arise during testing.

For more details on the ARM chip see the *Acorn RISC Machine family Data Manual*. VLSI Technology Inc. (1990) Prentice-Hall, Englewood Cliffs, NJ, USA: ISBN 0-13-781618-9.

Restrictions to the ARM instruction set

There are three main reasons for restricting the use of certain parts of the instruction set:

- **Dangerous instructions**

Such instructions can cause a program to fail unexpectedly, for example:

```
LDM R15,Rlist
```

uses PC+PSR as the base and so can cause an unexpected address exception.

- **Useless instructions**

It is better to reserve the instruction space occupied by existing ‘useless’ instructions for instruction expansion in future processors. For example:

```
MUL R15,Rm,Rs
```

only serves to scramble the PSR.

This category also includes ineffective instructions, such as a PC relative LDC/STC with writeback; the PC cannot be written back in these instructions, so the writeback bit is ineffective (and an attempt to use it should be flagged as an error).

Note: LDC/STC are instructions to load/store a coprocessor register from/to memory; since they are not supported by the BASIC assembler, they were not described in *Appendix A: ARM assembler*.

- **Instructions with undesirable side-effects**

It is hard to guarantee the side-effects of instructions across different processors. If, for example, the following is used:

```
LDR Rd,[R15,#expression]!
```

the PC writeback will produce different results on different types of processor.

Instructions and code sequences to avoid

The instructions and code sequences are split into a number of categories. Each category starts with an indication of which of the two main ARM variants (ARM2, ARM3) it applies to, and is followed by a recommendation or warning. The text then goes on to explain the conditions in more detail and to supply examples where appropriate.

Unless a program is being targeted **specifically** for a single version of the ARM processor family, all of these recommendations should be adhered to.

TSTP/TEQP/CMPP/CMNP: Changing mode

Applicability: ARM2

When the processor’s mode is changed by altering the mode bits in the PSR using a data processing operation, care must be taken not to access a banked register (R8-R14) in the following instruction. Accesses to the unbanked registers (R0-R7, R15) are safe.

The following instructions are affected, but note that mode changes can only be made when the processor is in a non-user mode:

```
TSTP Rn,Op2
TEQP Rn,Op2
MPP Rn,Op2
CMNP Rn,Op2
```

These are the only operations that change all the bits in the PSR (including the mode bits) without affecting the PC (thereby forcing a pipeline refill during which time the register bank select logic settles).

The following examples assume the processor starts in Supervisor mode:

- a) TEQP PC,#0
 MOV R0,R0
 ADD R0,R1,R13_usr
 Safe: NOP added between mode change and access to a banked register (R13_usr)
- b) TEQP PC,#0
 ADD R0,R1,R2
 Safe: No access made to a banked register
- c) TEQP PC,#0
 ADD R0,R1,R13_usr
 Fails: Data **not** read from Register R13_usr!

The safest default is always to add a NOP (e.g. MOV R0,R0) after a mode changing instruction; this will guarantee correct operation regardless of the code sequence following it.

LDM/STM: Forcing transfer of the user bank (Part 1)

Applicability: ARM2, ARM3

Do not use writeback when forcing user bank transfer in LDM/STM.

For STM instructions the S bit is redundant as the PSR is always stored with the PC whenever R15 is in the transfer list. In user mode programs the S bit is ignored, but in other modes it has a second interpretation; S=1 is used to force transfers to take values from the user register bank instead of from the current register bank. This is useful for saving the user state on process switches.

Similarly, in LDM instructions the S bit is redundant if R15 is not in the transfer list. In user mode programs, the S bit is ignored, but in non-usermode programs where R15 is not in the transfer list, S=1 is used to force loaded values to go to the user registers instead of the current register bank.

In both cases where the processor is in a non-user mode and transfer to or from the user bank is forced by setting the S bit, writeback of the base will also be to the user bank though the base will be fetched from the current bank. Therefore don't use writeback when forcing user bank transfer in LDM/STM.

The following examples assume the processor to be in a non-user mode and *Rlist* not to include R15:

STMxx Rn!, <i>Rlist</i>	Safe: Storing non-user registers with write back to the non-user base register
LDMxx Rn!, <i>Rlist</i>	Safe: Loading non-user registers with write back to the non-user base register
STMxx Rn, <i>Rlist</i> ^	Safe: Storing user registers, but no base write-back
STMxx Rn!, <i>Rlist</i> ^	Fails: Base fetched from non-user register, but written back into user register
LDMxx Rn!, <i>Rlist</i> ^	Fails: Base fetched from non-user register, but written back into user register

LDM: Forcing transfer of the user bank (Part 2)

Applicability: ARM2, ARM3

When loading user bank registers with an LDM in a non-user mode, care must be taken not to access a banked register (R8-R14) in the following instruction. Accesses to the unbanked registers (R0-R7,R15) are safe.

Because the register bank switches from user mode to non-user mode during the first cycle of the instruction following an LDM Rn,*Rlist*^, an attempt to access a banked register in that cycle may cause the wrong register to be accessed.

The following examples assume the processor to be in a non-user mode and *Rlist* not to include R15:

LDM Rn <i>Rlist</i> ^ ADD R0,R1,R2	Safe: Access to unbanked registers after LDM^
LDM Rn, <i>Rlist</i> ^ MOV R0,R0 ADD R0,R1,R13_svc	Safe: NOP inserted before banked register used following an LDM^
LDM Rn, <i>Rlist</i> ^ ADD R0,R1,R13_svc	Fails: Accessing a banked register immediately after an LDM^ returns the wrong data

<pre>ADR R14_svc, saveblock LDMIA R14_svc, {R0 - R14_usr}^ LDR R14_svc, [R14_svc,#15*4] MOVS PC, R14_svc (R14_svc)</pre>	<p>Fails: Banked base register used immediately after the LDM^</p>
<pre>ADR R14_svc, saveblock LDMIA R14_svc, {R0 - R14_usr}^ MOV R0,R0 LDR R14_svc, [R14_svc,#15*4] MOVS PC, R14_svc</pre>	<p>Safe: NOP inserted before banked register (R14_svc) used</p>

Note: The ARM2 and ARM3 processors **usually** give the expected result, but cannot be guaranteed to do so under all circumstances, therefore this code sequence should be avoided in future.

SWI/Undefined Instruction trap interaction

Applicability: ARM2

Care must be taken when writing an undefined instruction handler to allow for an unexpected call from a SWI instruction. The erroneous SWI call should be intercepted and redirected to the software interrupt handler.

The implementation of the CDP instruction on ARM2 may cause – under certain circumstances – a Software Interrupt (SWI) to take the Undefined Instruction trap if the SWI was the next instruction after the CDP. For example:

<pre>SIN F0 SWI &11</pre>	<p>Fails: ARM2 may take the undefined instruction trap instead of software interrupt trap.</p>
-------------------------------	---

All Undefined Instruction handler code should check the failed instruction to see if it is a SWI, and if so pass it over to the software interrupt handler by branching to the SWI hardware vector at address 8.

Note: CDP is a Coprocessor Data Operation instruction; since it is not supported by the BASIC assembler, it was not described in *Appendix A: ARM assembler*.

Undefined instruction/Prefetch abort trap interaction

Applicability: ARM2, ARM3

Care must be taken when writing the Prefetch abort trap handler to allow for an unexpected call due to an undefined instruction.

When an undefined instruction is fetched from the last word of a page, where the next page is absent from memory, the undefined instruction will cause the undefined instruction trap to be taken, and the following (aborted) instructions will cause a prefetch abort trap. One might expect the undefined instruction trap to be taken first, then the return to the succeeding code will cause the abort trap. In fact the prefetch abort has a higher priority than the undefined instruction trap, so the prefetch abort handler is entered before the undefined instruction trap, indicating a fault at the address of the undefined instruction (which is in a page which is actually present). A normal return from the prefetch abort handler (after loading the absent page) will cause the undefined instruction to execute and take the trap correctly. However the indicated page is already present, so the prefetch abort handler may simply return control, causing an infinite loop to be entered.

Therefore, the prefetch abort handler should check whether the indicated fault is in a page which is actually present, and if so it should suspect the above condition and pass control to the undefined instruction handler. This will restore the expected sequential nature of the execution sequence. A normal return from the undefined instruction handler will cause the next instruction to be fetched (which will abort), the prefetch abort handler will be re-entered (with an address pointing to the absent page), and execution can proceed normally.

Single instructions to avoid

Applicability: ARM2, ARM3

The following single instructions and code sequences should be avoided in writing any ARM code.

Any instruction that uses the ‘NV’ condition flag

Avoid using the NV (execute never) condition code:

*opcode*NV ...

i.e. any operation where *{cond}*= NV

By avoiding the use of the ‘NV’ condition code, 2^{28} instructions become free for future expansion.

Note: It is recommended that the instruction MOV R0,R0 be used as a general purpose NOP.

Data processing

Avoid using R15 in the Rs position of a data processing instruction:

MOV|MVN{*cond*}{S} Rd,Rm,*shiftname* R15

CMP|CMN|TEQ|TST{*cond*}{P} Rn,Rm,*shiftname* R15

ADC|ADD|SBC...|EOR{cond}{S} Rd,Rn,shiftname R15

Shifting a register by an amount dependent upon the code position should be avoided.

Multiply and multiply-accumulate

Do not specify R15 as the destination register as only the PSR will be affected by the result of the operation:

MUL{cond}{S} R15,Rm,Rs
MLA{cond}{S} R15,Rm,Rs,Rn

Do not use the same register in the Rd and Rm positions, as the result of the operation will be incorrect:

MUL{cond}{S} Rd,Rd,Rs
MLA{cond}{S} Rd,Rd,Rs

Single data transfer

Do not use a PC relative load or store with base writeback as the effects may vary in future processors:

LDR|STR{cond}{B}{T} Rd,[R15,#expression]!
LDR|STR{cond}{B}{T} Rd,[R15,-]Rm{,shift}!
LDR|STR{cond}{B}{T} Rd,[R15],#expression
LDR|STR{cond}{B}{T} Rd,[R15,-]Rm{,shift}

Note: It is safe to use pre-indexed PC relative loads and stores **without** base writeback.

Avoid using R15 as the register offset (Rm) in single data transfers as the value used will be PC+PSR which can lead to address exceptions:

LDR|STR{cond}{B}{T} Rd,[Rn,-]R15{,shift}(!)
LDR|STR{cond}{B}{T} Rd,[Rn,-]R15{,shift}

A byte load or store operation on R15 must not be specified, as R15 contains the PC, and should always be treated as a 32 bit quantity:

LDR|STR{cond}B{T} R15,Address

A post-indexed LDR|STR where Rm=Rn must not be used (this instruction is very difficult for the abort handler to unwind when late aborts are configured – which do not prevent base writeback):

LDR|STR{cond}{B}{T} Rd,[Rn,-]Rn{,shift}

Do not use the same register in the Rd and Rm positions of an LDR which specifies (or implies) base writeback; such an instruction is ambiguous, as it is not clear whether the end value in the register should be the loaded data or the updated base:

```
LDR{cond}{B}{T} Rn,[Rn,#expression]!
LDR{cond}{B}{T} Rn,[Rn,{-}Rm{,shift}]!

LDR{cond}{B}{T} Rn,[Rn],#expression
LDR{cond}{B}{T} Rn,[Rn],{-}Rm{,shift}
```

Block data transfer

Do not specify base writeback when forcing user mode block data transfer as the writeback may target the wrong register:

```
STM{cond}<FDIED...!DB> Rn!,Rlist^
LDM{cond}<FDIED...!DB> Rn!,Rlist^
```

where *Rlist* does not include R15.

Note: The instruction:

```
LDM{cond}<FDIED...!DB> Rn!,<Rlist,R15>^
```

does **not** force user mode data transfer, and can be used safely.

Do not perform a PC relative block data transfer, as the PC+PSR is used to form the base address which can lead to address exceptions:

```
LDMISTM{cond}<FDIED...!DB> R15{!},Rlist{^}
```

Single data swap

Do not perform a PC relative swap as its behaviour may change in the future:

```
SWP{cond}{B} Rd,Rm,[R15]
```

Avoid specifying R15 as the source or destination register:

```
SWP{cond}{B} R15,Rm,[Rn]
SWP{cond}{B} Rd,R15,[Rn]
```

Note: SWP is a Single Data Swap instruction, typically used to implement semaphores, and introduced in the ARM3; since it is not supported by the BASIC assembler, it was not described in *Appendix A: ARM assembler*.

Coprocessor data transfers

When performing a PC relative coprocessor data transfer, writeback to R15 is prevented so the W bit should not be set:

```
LDCISTC{cond}{L} CP#,CRd,[R15]!
LDCISTC{cond}{L} CP#,CRd,[R15,#expression]!
LDCISTC{cond}{L} CP#,CRd,[R15]#expression!
```

Undefined instructions

ARM2 has two undefined instructions, and ARM3 has only one (the other ARM2 undefined instruction has been defined as the Single data swap operation).

Undefined instructions should not be used in programs, as they may be defined as a new operation in future ARM variants.

Register access after an in-line mode change

Care must be taken not to access a banked register (R8-R14) in the cycle following an in-line mode change. Thus the following code sequences should be avoided:

- 1 TSTP|TEQP|CMPP|CMNP{*cond*} Rn,Op2
- 2 any instruction that uses R8-R14 in its first cycle.

Register access after an LDM that forces user mode data transfer

The banked registers (R8-R14) should not be accessed in the cycle immediately after an LDM that forces user mode data transfer. Thus the following code sequence should be avoided:

- 1 LDM{*cond*}<FD|ED...|DB> Rn,Rlist^
where *Rlist* does **not** include R15
- 2 any instruction that uses R8-R14 in its first cycle.

Other points to note

This section highlights some obscure cases of ARM operation which should be borne in mind when writing code.

Use of R15

Applicability: ARM2, ARM3

Warning: When the PC is used as a destination, operand, base or shift register, different results will be obtained depending on the instruction and the exact usage of R15.

Full details of the value derived from or written into R15+PSR for each instruction class is given in the *Acorn RISC Machine family Data Manual*. Care must be taken when using R15 because small changes in the instruction can yield significantly different results. For example, consider data operations of the type:-

```
opcode{cond}{S} Rd,Rn,Rm
or opcode{cond}{S} Rd,Rn,Rm,shiftname Rs
```

- When R15 is used in the Rm position, it will give the value of the PC together with the PSR flags.
- When R15 is used in the Rn or Rs positions, it will give the value of the PC without the PSR flags (PSR bits replaced by zeros).

```
MOV R0,#0
ORR R1,R0,R15      ; R1:=PC+PSR  (bits 31:26,1:0 reflect PSR flags)
ORR R2,R15,R0      ; R2:=PC      (bits 31:26,1:0 set to zero)
```

Note: The relevant instruction description in the *ARM Acorn RISC Machine family Data Manual* should be consulted for full details of the behaviour of R15.

STM: Inclusion of the base in the register list

Applicability: ARM2, ARM3

Warning: In the case of a STM with writeback that includes the base register in the register list, the value of the base register stored depends upon its position in the register list.

During an STM, the first register is written out at the start of the second cycle of the instruction. When writeback is specified, the base is written back at the end of the second cycle. An STM which includes storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, it will store the modified value.

For example:

```
MOV R5,#&1000
STMIA R5!,{R5-R6}      ; Stores value of R5=&1000

MOV R5,#&1000
STMIA R5!,{R4-R5}      ; Stores value of R5=&1008
```

MUL/MLA: Register restrictions

Applicability: ARM2, ARM3

```
Given      MUL Rd,Rm,Rs
or         MLA Rd,Rm,Rs,Rn

Then       Rd & Rm must be different registers
           Rd must not be R15
```

Due to the way the Booth's algorithm has been implemented, certain combinations of operand registers should be avoided. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register (Rd) should not be the same as the Rm operand register, as Rd is used to hold intermediate values and Rm is used repeatedly during the multiply. A MUL will give a zero result if Rm=Rd, and a MLA will give a meaningless result.

The destination register (Rd) should also not be R15. R15 is protected from modification by these instructions, so the instruction will have no effect, except that it will put meaningless values in the PSR flags if the S bit is set.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

LDM/STM: Address Exceptions

Applicability: ARM2, ARM3

Warning: Illegal addresses formed during a LDM or STM operation will not cause an address exception.

Only the address of the first transfer of a LDM or STM is checked for an address exception; if subsequent addresses over-flow or under-flow into illegal address space they will be truncated to 26 bits but will not cause an address exception trap.

The following examples assume the processor is in a non-user mode and MEMC is being accessed:

```
MOV R0,#&04000000      ; R0=&04000000
STMIA R0,{R1-R2}      ; Address exception reported
                       : (base address illegal)

MOV R0,#&04000000
SUB R0,R0,#4           ; R0=&03FFFFFFC
STMIA R0,{R1-R2}      ; No address exception reported
                       : (base address legal)
                       ; code will overwrite data at address &00000000
```

Note: The exact behaviour of the system depends upon the memory manager to which the processor is attached; in some cases, the wraparound may be detected and the instruction aborted.

LDC/STC: Address Exceptions

Applicability: ARM2, ARM3

Warning: Illegal addresses formed during a LDC or STC operation will not cause an address exception (affects LDF/STF).

The coprocessor data transfer operations act like STM and LDM with the processor generating the addresses and the coprocessor supplying/reading the data. As with LDM/STM, only the address of the first transfer of a LDC or STC is checked for an address exception; if subsequent addresses over-flow or under-flow into illegal address space they will be truncated to 26 bits but will not cause an address exception trap.

Note that the floating point LDF/STF instructions are forms of LDC and STC.

The following examples assume the processor is in a non-user mode and MEMC is being accessed:

```
MOV R0,#&04000000      ; R0=&04000000
STC CP1,CR0,[R0]      ; Address exception reported
                       :   (base address illegal)

MOV R0,#&04000000
SUB R0,R0,#4           ; R0=&03FFFFFFC
STFD F0,[R0]          ; No address exception reported
                       :   (base address legal)
                       ; code will overwrite data at address &00000000
```

Note: The exact behaviour of the system depends upon the memory manager to which the processor is attached; in some cases, the wraparound may be detected and the instruction aborted.

LDC: Data transfers to a coprocessor fetch more data than expected

Applicability: ARM3

Data to be transferred to a coprocessor with the LDC instruction should never be placed in the last word of an addressable chunk of memory, nor in the word of memory immediately preceding a read-sensitive memory location.

Due to the pipelining introduced into the ARM3 coprocessor interface, an LDC operation will cause one extra word of data to be fetched from the internal cache or external memory by ARM3 and then discarded; if the extra data is fetched from an area of external memory marked as cacheable, a whole line of data will be fetched and placed in the cache.

A particular case in point is that an LDC whose data ends at the last word of a memory page will load and then discard the first word (and hence the first cache line) of the next page. A minor effect of this is that it may occasionally cause an unnecessary page swap in a virtual memory system. The major effect of it is that (whether in a virtual memory system or not), the data for an LDC should never be placed in the last word of an addressable chunk of memory: the LDC will attempt to read the immediately following non-existent location and thus produce a memory fault.

The following example assumes the processor is in a non-user mode, FPU hardware is attached and MEMC is being accessed:

```
MOV R13,#&03000000      ; R13=Address of I/O space
STFD F0,[R13,#-8]!     ; Store F.P. register 0 at top of physical memory
                        ; (two words of data transferred)
LDFD F1,[R13],#8       ; Load F.P. register 1 from top of physical
                        ; memory, but three words of data are
                        ; transferred, and the third access will read
                        ; from I/O space which may be read sensitive
```

Static ARM problems

The static ARM is a variant of the ARM processor designed for low power consumption, that is built using static CMOS technology. (The difference between it and the standard ARM is similar to that between SRAM and DRAM.)

The static ARM exhibits different behaviour to ARM2 and ARM3 when executing a PC relative LDR with base writeback. This class of instruction has very limited application, so the discrepancy should not be a problem, but if you wish to use any of the following instructions in your code you are advised to contact Acorn Computers.

```
LDR Rd,[PC,#expression]!
LDR Rd,[PC],#expression
LDR Rd,[PC,{-}Rm{,shift}]!
LDR Rd,[PC],{-}Rm{,shift}
```

Note: A PC relative LDR **without** writeback works exactly as expected.

Provided that this instruction class is unused, it is likely that writeback to the PC on LDR and STR will be disabled completely in the future. The fewer incidental ways there are to modify the PC the better.

Unexpected Static ARM2 behaviour when executing a PC relative LDR with writeback

The instructions affected are:-

- LDR Rd,[PC,#*expression*]!

-
- LDR Rd,[PC],#expression

Case 1: LDR Rd,[PC,#expression]!

Expected result: $Rd \leftarrow (PC+8+expression)$
 $PC \leftarrow PC+8+expression$
 ...so execution continues from $PC+8+expression$

Actual ARM2 result: $Rd \leftarrow Rd$ {no change}
 $PC \leftarrow PC+8+expression+4$
 ...so execution continues from $PC+12+expression$

Case 2: LDR Rd,[PC],#expression

Expected result: $Rd \leftarrow (PC+8)$
 $PC \leftarrow PC+8+expression$
 ...so execution continues from $PC+8+expression$

Actual ARM2 result: $Rd \leftarrow Rd$ {no change}
 $PC \leftarrow PC+8+expression+4$
 ...so execution continues from $PC+12+expression$

Appendix C: ARM procedure call standard

This appendix relates to the implementation of compiler code-generators and language run-time library kernels for the Acorn RISC Machine (ARM) but is also a useful reference when interworking assembly language with high level language code.

The reader should be familiar with the ARM's instruction set, floating-point instruction set and assembler syntax 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 in this appendix are the procedure call and stack disciplines. These disciplines are observed by Acorn's C language implementation for the ARM and, eventually, will be observed by other high level language 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 appendix are several examples of the usage of this standard, together with suggestions for generating effective code for the ARM.

The purpose of APCS

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). 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 appendix 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 dereferencing 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 2:1 to 5:1.)
- 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

This section defines the standard.

Register names

The ARM has 16 visible general registers and 8 floating-point registers. 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 of the procedure call standard* on page 4-407.

The register names and fixed bindings are given immediately below.

General Registers

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
```

Then the call-frame registers, the bindings of which vary (see the section entitled *Defined bindings of the procedure call standard* on page 4-407 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
```

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
```

In the obsolete APCS-A register bindings described below, sp is bound to r12; in all other APCS bindings, sp is bound to r13.

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 must 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.

The floating-point registers are:

<code>f0</code>	<code>FN 0</code>	; floating point result (or 1st FP argument)
<code>f1</code>	<code>FN 1</code>	; floating point scratch register (or 2nd FP arg)
<code>f2</code>	<code>FN 2</code>	; floating point scratch register (or 3rd FP arg)
<code>f3</code>	<code>FN 3</code>	; floating point scratch register (or 4th FP arg)
<code>f4</code>	<code>FN 4</code>	; floating point preserved register
<code>f5</code>	<code>FN 5</code>	; floating point preserved register
<code>f6</code>	<code>FN 6</code>	; floating point preserved register
<code>f7</code>	<code>FN 7</code>	; floating point preserved register

Data representation and argument passing

The APCS 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.

Register usage and argument passing to external procedures

Control Arrival

We consider the passing of $N (\geq 0)$ actual argument words to a procedure which expects to receive either exactly N argument words or a variable number $V (\geq 1)$ 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):

```
arg1 is in a1
arg2 is in a2
arg3 is in a3
arg4 is in a4
for all  $I \geq 5$ ,  $\text{arg}I$  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 four.

lr contains the $\text{pc}+\text{psw}$ value that should be restored into r15 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.

$sp \geq SP_LWM + 256$.

This condition guarantees that a stack extension procedure, if used, will have a reasonable amount – 256 bytes – of work space available to it, probably sufficient to call two or three procedure invocations further.

Control Return

At the instant when the return link value for a procedure call is placed in the pc+psw, the following statements shall be true:

fp, sp, sl, v1-v6, and f4-f7 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 a1. If the procedure returns a floating-point result, FPR, then FPR shall be in f0.

Notes

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 K&R 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. However, 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.

The flag values from lr at the instant of entry must be restored; 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
MOVLT   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. 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 large, 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 Arthur/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 section entitled *Defined bindings of the procedure call standard* on page 4-407 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, 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 it 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).

The 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:

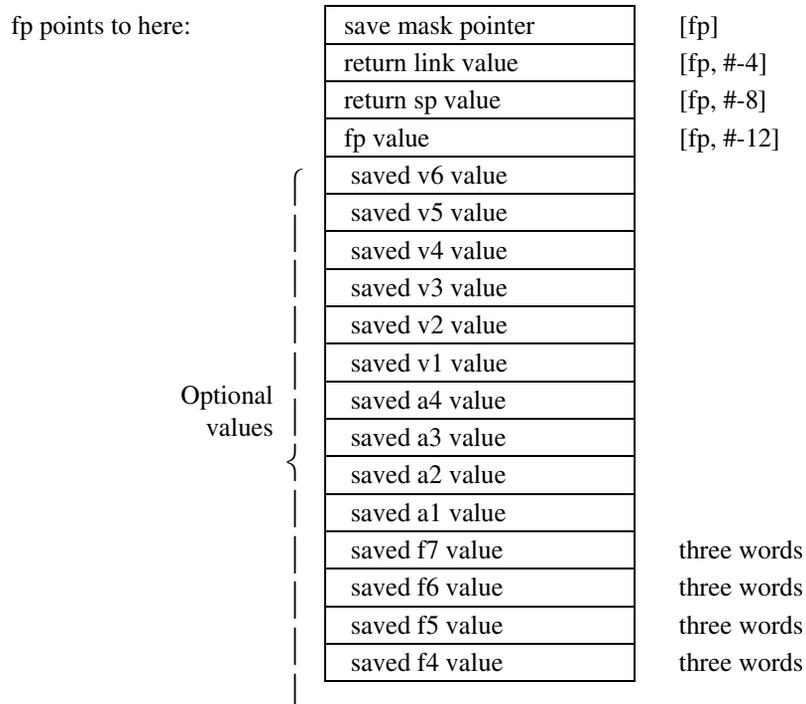


Figure 88.1 Stack backtrace data structure

This picture shows between four and 26 words of store, with those words higher on the page being at higher addresses in memory. The presence of any of the optional values 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 x 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]! ; 1110 1101 0110 1101 0111 0001 0000 0011
STFE f6, [sp, #-12]! ; 1110 1101 0110 1101 0110 0001 0000 0011
STFE f5, [sp, #-12]! ; 1110 1101 0110 1101 0101 0001 0000 0011
STFE f4, [sp, #-12]! ; 1110 1101 0110 1101 0100 0001 0000 0011
```

!

Any or all of these instructions may be missing, and any deviation from this order or any other instruction terminates the sequence.

(A 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.)

The bit patterns given are for APCS-R/APCS-U register bindings. In the obsolete APCS-A bindings, the bit indicated by ! 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. The 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 as 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 of the procedure call standard

APCS-R and APCS-U: The RISC OS and RISC iX PCSs

These bindings of the APCS 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:

<code>sl</code>	RN	10	; stack limit / stack chunk handle ; unused by RISC iX applications
<code>fp</code>	RN	11	; frame pointer
<code>ip</code>	RN	12	; used as temporary workspace
<code>sp</code>	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 relabelling 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.
- No use is made of the ordering of the four call-frame registers (eg in order to load/save `fp` or `sp` from a full register save).

APCS-R: Constraints on `sl` (For RISC OS applications and modules)

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**.

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.

APCS-U: Constraints on *sl* (For RISC iX applications and RISC iX kernels)

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 by Acorn.

APCS-A: The obsolete Arthur application PCS

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:

<i>sl</i>	RN	13	; stack limit/stack chunk handle
<i>fp</i>	RN	10	; frame pointer
<i>ip</i>	RN	11	; used as temporary workspace
<i>sp</i>	RN	12	; lower end of current stack frame

(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:

<code>sl</code>	RN	12	; stack limit/stack chunk handle
<code>fp</code>	RN	10	; frame pointer
<code>ip</code>	RN	11	; used as temporary workspace
<code>sp</code>	RN	13	; lower end of current stack frame

This binding of APCS is unlikely to be used (by Acorn).

Further Restrictions in SVC Mode and IRQ Mode

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 return to the saved `r14_irq` (which also 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 generate code to save and restore `r14` in-line, around the SWI, unless it is known that the code will not be executed 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 (causing a data abort). 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 second or third 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).

Examples from Acorn language implementations

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
      STMFD 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 do:

```
      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 four arguments it can push all of them onto the stack at the same time as saving its old fp and its return address (see the example above); arguments are then saved contiguously in memory with arg1 having the lowest address. A function that expects more than four arguments has code at its head as follows:

```
MOV    ip, sp
STMFD  sp!, {a1, a2, a3, a4} ; put arg1-4 below stacked args
STMFD  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   lx$stack_overflow!
...
...
LDMEA  fp, {v1, v2, fp, sp, pc}^ ; restore register vars & return
```

The store of the argument registers shown here is not mandated by APCS and can often be omitted. It is useful in support of debuggers and run-time trace-back code and required if the address of an argument is taken.

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 language implementations

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 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/Topexpress Arthur/RISC OS Fortran-77 compiler violates the APCS in a number of ways that preclude inter-working with C, except via assembler veneers. This may be changed in future releases of the Fortran-77 product.

Pascal

The Acorn/3L Arthur/RISC OS ISO-Pascal compiler violates the APCS in a number of ways that preclude inter-working with C, except via assembler veneers. This may be changed in future releases of the ISO-Pascal product.

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 appendix is not intended as a general guide to the writing of code-generators, but it is worth highlighting 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 10% (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
    MOVL  b, a1
```

In the case of a **leaf** or **fast** procedure – 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 `get_a` 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
     MOVSL  pc, lr ; and if appropriate, handle quickly!
     ; else, form a stack frame and handle the rest as normal code.
     MOV  ip, sp
     STMDB sp!, {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
     LDMEA 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 not performed by the C compiler.

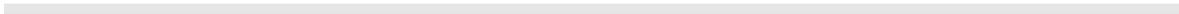
Finally, it is often worth applying the tail call optimisation, especially to procedures which need to save no registers. For example, the code fragment:

```
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, LSR #2 ; 1S
        MOV  a1, #1073741824 ; 1S
        B   primitive_alloc ; 1N+2S = 4S
```

This avoids saving and restoring the call-frame registers and minimises the cost of interface ‘sugaring’ procedures. This saves five instructions and, on a 4/8MHz ARM, reduces the cost of the malloc sugar from 24S to 7S.



This appendix defines three file formats used to store processed code and the format of debugging data used by debuggers:

- AOF – Arm Object Format
- ALF – Acorn Library Format
- AIF – RISC OS Application Image Format
- ASD – ARM Symbolic Debugging Format.

Language processors such as CC and ObjAsm generate processed code output as AOF files. An ALF file is a collection of AOF files constructed from a set of AOF files by the LibFile tool. The Link tool accepts a set of AOF and ALF files as input, and by default produces an executable program file as output in AIF.

Terminology

Throughout this appendix 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 (ie 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 (ie 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 appendix 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 AOF and ALF files

An object or library 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 and library file formats are themselves 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
- string table.

The library file format defines four chunks:

- directory
- time-stamp
- version
- data.

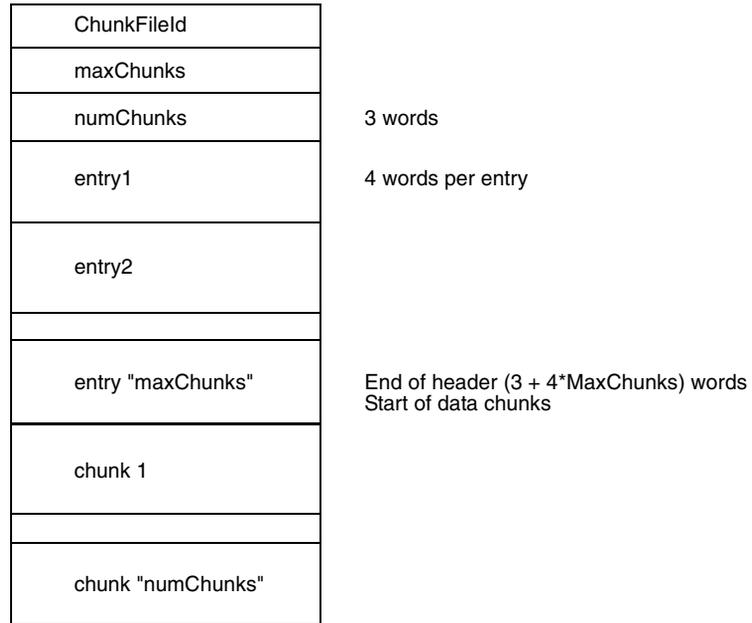
There may be many data chunks in a library.

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 file contains
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 later. For ALF files, the first part is LIB_.

AOF

ARM object format files are output by language processors such as CC and ObjAsm.

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 by 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	
Version Id	
Number of areas	
Number of Symbols	
Entry Address area	
Entry Address Offset	6 words in the fixed part
1st Area Header	5 words per area header
2nd Area Header	
nth Area Header	(6 + 5*Number of Areas) words in the AOF header

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 nth 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 variable)
zeros	AT	AL	
Area size			5 words in total
Number of relocations			
Unused - must be zero			

Area name

Each name in an object file is encoded as an offset into the string table, which 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.

Link 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 Link. 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 the section entitled *Linker defined symbols* on page 4-431).

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, Link 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 the section on Bit 5, below).

Note: Whether or not a zero-initialised area is re-zeroed if the image is re-entered is a property of Link and the relevant image format. The definition of AOF neither requires nor precludes re-zeroing.

Bit 5

This bit specifies that the area is read-only. Link groups read-only areas together so that they may be write protected at run-time, hardware permitting. Code areas and debugging tables should have this bit set. The setting of this bit is incompatible with the setting of bit 4.

Bit 6

This bit must be set to 0.

Bit 7

This bit specifies that the area contains symbolic debugging tables. Link groups these areas together so they can be accessed as a single contiguous chunk at run-time. It is usual for debugging tables to be read-only and, therefore, to have bit 5 set too. If bit 7 is set, bit 1 is ignored.

Area size

This field specifies the size of the area in bytes, which must be a multiple of 4. Unless the Not Initialised bit (bit 4) is set in the area attributes, there must be this number of bytes for this area in the OBJ_AREA chunk.

Number of relocations

This specifies the number of relocation directives which apply to this area.

Format of the areas chunk

The areas chunk (OBJ_AREA) contains the actual areas (code, data, zero- initialised data, debugging data, etc.) plus any associated relocation information. Its chunkId is OBJ_AREA. Both an area's contents and its relocation data must be word-aligned. Graphically, the layout of the areas chunk is:

Area 1
Area 1 relocation
Area n
Area n relocation

An area is simply a sequence of byte values, the order following that of the addressing rules of the ARM, that is the least significant byte of a word is first. An area is followed by its associated relocation table (if any). An area is either completely initialised by the values from the file or not initialised at all (ie it is initialised to zero in any loaded program image, as specified by bit 4 of the area attributes).

Relocation directives

If no relocation is specified, the value of a byte/half word/word in the preceding area is exactly the value that will appear in the final image.

Bytes and half words 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 1.50 and later.

Relocation can take two basic forms: *Additive* and *PCRelative*.

Additive relocation specifies the modification of a byte/half word/word, typically containing a data value (ie 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 Symbol Identification field (SID) 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. Link 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 version 1.50 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 Link, 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.

Link itself may generate type-2, PC-relative, internal relocation directives during the process of partially linking a set of object modules.

Format of Type 1 relocation directives

Diagrammatically:

Offset				
O	A	R	FT	SID

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	half word
10	word
11	<i>illegal value</i>

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 by 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 directives* on page 4-426.

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	

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, Link 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 Link 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 (ie when attempting to resolve unresolved references, Link will match this symbol to references from other object files).
- 00** Reserved by 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, Link 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

This bit is only meaningful if the symbol is a defining, external occurrence (ie 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.

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.

Bit 6

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). Link 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.

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.

All common symbols for which there is no matching common area (reference or definition) are collected into an anonymous linker pseudo-area.

Value

This field is only meaningful if the symbol is a defining occurrence (ie bit 0 of AT set) or a common symbol (ie 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.

Area name

This field is only meaningful if the symbol is not absolute (ie if bit 2 of AT is unset) and the symbol is a defining occurrence (ie 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.

String table chunk (OBJ_STRT)

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 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.

Identification chunk (OBJ_IDFN)

This chunk should contain a printable character string (characters in the range [32 - 126]), 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 by Acorn.

None of these symbols may be redefined. The pre-defined symbols are:

Image\$\$RO\$\$Base	Address and limit of the Read-Only section
Image\$\$RO\$\$Limit	of the image.
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 (created from areas having bit 4 of their area attributes set and from common symbols which match no area name).

If a section is absent, the Base and Limit values are equal but unpredictable.

Image\$\$RO\$\$Base	includes any image header prepended by Link.
---------------------	--

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 Link.

Common area symbols

For each common area, Link 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 subsections 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 obsolete Dbug (DDT has Dbug's functionality and uses 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 by 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 RISC OS linker and therefore had no utility. Link in any case allows 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 Link. And use was hedged around with curious restrictions). Accordingly, the definition of fragmented areas is withdrawn.

ALF

ALF is the format of linkable libraries (such as the C RISC OS library RISC_OSLib).

Library file format types

There are two library file formats described here, termed *new-style* and *old-style*. Link can read both formats, though no tool will actually generate an old-style library.

Currently, only the Acorn/Topexpress Fortran-77 compiler generates old-style libraries (which it does instead of generating AOF object files). Link handles these libraries specially, including every member in the output image unless explicitly instructed otherwise.

Old-style libraries are obsolescent and should no longer be generated.

Library file chunks

Each piece of a library file is stored in a separate, identifiable, chunk, named as follows:

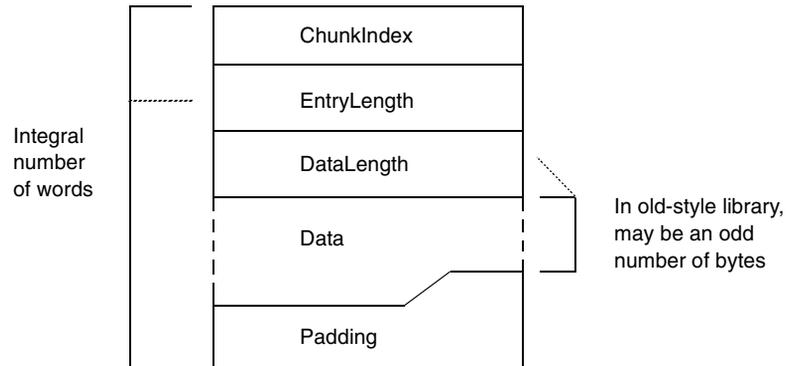
Chunk	Chunk Name	
Directory	LIB_DIRY	
Time-stamp	LIB_TIME	
Version	LIB_VSRN	– new-style libraries only
Data	LIB_DATA	
Symbol table	OFL_SYMT	– object code libraries only
Time-stamp	OFL_TIME	– object code libraries only

There may be many LIB_DATA chunks in a library, one for each library member.

LIB_DIRY

The LIB_DIRY chunk contains a directory of all modules in the library each of which is stored in a LIB_DATA chunk. The directory size is fixed when the library is created. The directory consists of a sequence of variable length entries, each an integral number of words long. The number of directory entries is determined by the size of the LIB_DIRY chunk.

This is shown pictorially in the following diagram:



ChunkIndex

The **ChunkIndex** is a 0 origin index within the chunk file header of the corresponding **LIB_DATA** chunk. The **LIB_DATA** chunk entry gives the offset and size of the library module in the library file. A **ChunkIndex** of 0 means the directory entry is not in use.

EntryLength

The number of bytes in this **LIB_DIRY** entry, always a multiple of 4.

DataLength

The number of bytes used in the **Data** section of this **LIB_DIRY** entry. This need not be a multiple of 4, though it always is in new-style libraries.

Data

The data section consists of a 0 terminated string followed by any other information relevant to the library module. Strings should contain only ISO-8859 non-control characters (ie codes [0-31], 127 and 128+[0-31] are excluded). The string is the name used by the library management tools to identify this library module. Typically this is the name of the file from which the library member was created.

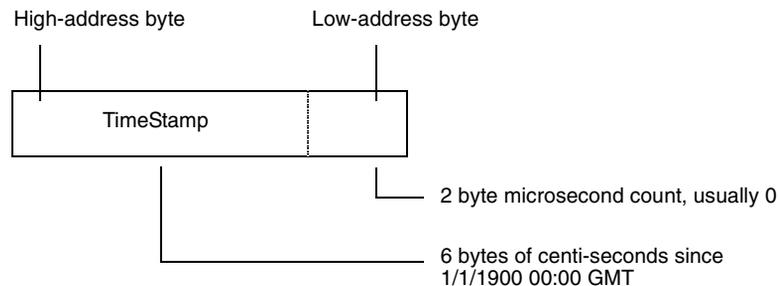
In new-style libraries, an 8-byte, word-aligned time-stamp follows the member name. The format of this time-stamp is described in the section entitled *LIB_TIME* on page 4-436. Its value is (an encoded version of) the time-stamp (ie the last modified time) of the file from which the library member was created.

Applications which create libraries or library members should ensure that the LIB_DIRY entries they create contain valid time-stamps. Applications which read LIB_DIRY entries should not rely on any data beyond the end of the name-string being present unless the difference between the DataLength field and the name-string length allows for it. Even then, the contents of a time-stamp should be treated cautiously and not assumed to be sensible.

Applications which write LIB_DIRY or OFL_SYMT entries should ensure that padding is done with NUL (0) bytes; applications which read LIB_DIRY or OFL_SYMT entries should make no assumptions about the values of padding bytes beyond the first, string-terminating NUL byte.

LIB_TIME

The LIB_TIME chunk contains a 64 bit time-stamp recording when the library was last modified, in the following format:



LIB_VSRN

In new-style libraries, this chunk contains a 4-byte version number. The current version number is 1. Old-style libraries do not contain this chunk.

LIB_DATA

A LIB_DATA chunk contains one of the library members indexed by the LIB_DIRY chunk. No interpretation is placed on the contents of a member by the library management tools. A member could itself be a file in chunk file format or even another library.

Object code libraries

An object code library is a library file whose members are files in AOF. All libraries you are likely to use with the DDE are object code libraries.

Additional information is stored in two extra chunks, `OFL_SYMT` and `OFL_TIME`.

`OFL_SYMT` contains an entry for each external symbol defined by members of the library, together with the index of the chunk containing the member defining that symbol.

The `OFL_SYMT` chunk has exactly the same format as the `LIB_DIRY` chunk except that the Data section of each entry contains only a string, the name of an external symbol (and between 1 and 4 bytes of NUL padding). `OFL_SYMT` entries do not contain time-stamps.

The `OFL_TIME` chunk records when the `OFL_SYMT` chunk was last modified and has the same format as the `LIB_TIME` chunk (see above).

AIF

AIF is the format of executable program files produced by linking AOF files. Example AIF files are !RunImage files of applications coded in C or assembler.

Properties of AIF

- An AIF image is loaded into memory at its load address and entered at its first word (compatible with old-style Arthur/Brazil ADFS images).
- An AIF image may be compressed and can be self-decompressing (to support faster loading from floppy discs, and better use of floppy-disc space).
- If created with suitable linker options, an AIF image may relocate itself at load time. Self-relocation is supported in two, distinct senses:
 - One-time Position-Independence: A relocatable image can be loaded at any address (not just its load address) and will execute there (compatible with version 0.03 of AIF).
 - Specified Working Space Relocation: A suitably created relocatable image will copy itself from where it is loaded to the high address end of applications memory, leaving space above the copied image as noted in the AIF header (see below).

In addition, similar relocation code and similar linker options support many-time position independence of RISC OS Relocatable Modules.

- AIF images support being debugged by the Desktop Debugging Tool (DDT), for C, assembler and other languages. Version 0.04 of AIF (and later) supports debugging at the symbolic assembler level (hitherto done by Dbug). Low-level and source-level debugging support are orthogonal (capabilities of debuggers notwithstanding, both, either, or neither kind of debugging support may be present in an AIF image).

Debugging tables have the property that all references from them to code and data (if any) are in the form of relocatable addresses. After loading an image at its load address these values are effectively absolute. All references between debugger table entries are in the form of offsets from the beginning of the debugging data area. Thus, following relocation of a whole image, the debugging data area itself is position independent and can be copied by the debugger.

Layout of an AIF image

The layout of an AIF image is as follows:

Header	
Compressed image	
Decompression data	This data is position-independent
Decompression code	This code is position-independent

The header is small, fixed in size, and described below. In a compressed AIF image, the header is NOT compressed.

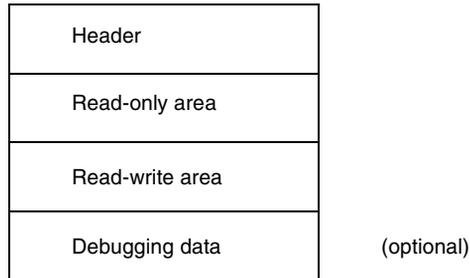
Once an image has been decompressed – or if it is uncompressed in the first place – it has the following layout:

Header	
Read-only area	
Read-write area	
Debugging data	(optional)
Self-relocation code	Must be position-independent
Relocation list	List of words to relocate, terminated by -1

Debugging data are absent unless the image has been linked appropriately and, in the case of source-level debugging, unless the constituent components of the image have been compiled appropriately.

The relocation list is a list of byte offsets from the beginning of the AIF header, of words to be relocated, followed by a word containing -1. The relocation of non-word values is not supported.

After the execution of the self-relocation code – or if the image is not self-relocating – the image has the following layout:



At this stage a debugger is expected to copy the debugging data (if present) somewhere safe, otherwise they will be overwritten by the zero-initialised data and/or the heap/stack data of the program. A debugger can seize control at the appropriate moment by copying, then modifying, the third word of the AIF header (see below).

AIF header layout

BL DecompressedCode	BLNV 0 if the image is not compressed
BL SelfRelocCode	BLNV 0 if the image is not self-relocating
BL ZeroInitCode	BLNV 0 if the image has none
BL ImageEntryPoint	BL to make header addressable via R14
SWI OS_Exit	Just in case silly enough to return
Image ReadOnly size	Includes header size and any padding Exact size - a multiple of 4 bytes
Image ReadWrite size	Exact size - a multiple of 4 bytes
Image Debug size	Exact size - a multiple of 4 bytes
Image zero-init size	Exact size - a multiple of 4 bytes
Image debug type	0,1,2 or 3 (see below)
Image base	Address of the AIF header - set by link
Work space	Min work space - in bytes - to be reserved by a self-moving relocatable image
Four reserved words (0)	
Zero-init code (16 words)	Header is 32 words long

BL is used everywhere to make the header addressable via R14 (but beware the PSR bits) in a position-independent manner and to ensure that the header will be position-independent.

It is required that an image be re-entrant at its first instruction. Therefore, after decompression, the decompression code must reset the first word of the header to BLNV 0. Similarly, following self-relocation, the second word of the header must be reset to BLNV 0. This causes no additional problems with the read-only nature of the code segment – both decompression and relocation code must write to it anyway. So, on systems with memory protection, both the decompression code and the self-relocation code must be bracketed by system calls to change the access status of the read-only section (first to writable, then back to read-only).

The image debug type has the following meaning:

- 0: No debugging data are present.
- 1: Low-level debugging data are present.
- 2: Source level (ASD) debugging data are present.
- 3: 1 and 2 are present together.

All other values are reserved by Acorn.

Zero-initialisation code

The Zero-initialisation code is as follows:

```

BIC    R11,    LR,    #&FC000003 ; clear status bits -> header + &C
ADD    R11,    R11,    #8 ; -> Image ReadOnly size
LDMIA  R11,    {R0,R1,R2,R3} ; various sizes
CMPS   R3,    #0
MOVLES PC,    LR ; nothing to do
SUB    R11,    R11,    #&14 ; image base
ADD    R11,    R11,    R0 ; + RO size
ADD    R11,    R11,    R1 ; + RW size = base of 0-init area
MOV    R0,    #0
MOV    R1,    #0
MOV    R2,    #0
MOV    R4,    #0
ZeroLoop
STMIA  R11!,   {R0,R1,R2,R4}
SUBS   R3,    R3,    #16
BGT    ZeroLoop
MOVS   PC,    LR ; 16 words in total.
```

Relationship between header sizes and linker pre-defined symbols

AIFHeader.ImageBase	= Image\$\$RO\$\$Base
AIFHeader.ImageBase + AIFHeader.ROSize	= Image\$\$RW\$\$Base
AIFHeader.ImageBase + AIFHeader.ROSize + AIFHeader.RWSize	= Image\$\$ZI\$\$Base
AIFHeader.ImageBase + AIFHeader.ROSize + AIFHeader.RWSize + AIFHeader.ZeroInitSize	= Image\$\$RW\$\$Limit

Self relocation

Two kinds of self-relocation are supported by AIF and one by AMF; for completeness, all three are described here.

One-time position independence is supported by relocatable AIF images. Many-time position independence is required for AMF Relocatable Modules. And only AIF images can self-move to a location which leaves a requested amount of workspace.

Why are there three different kinds of self-relocation?

- The rules for constructing RISC OS applications do not forbid acquired position-dependence. Once an application has begun to run, it is not, in general, possible to move it, as it isn't possible to find all the data locations which are being used as position-dependent pointers. So, AIF images can be relocated only once. Afterwards, the relocation table is over-written by the application's zero-initialised data, heap, or stack.
- In contrast, the rules for constructing a RISC OS Relocatable Modules (RM) require that it be prepared to shut itself down, be moved in memory, and start itself up again. Shut-down and start-up are notified to a RM by special service calls to it. Clearly, a RM must be relocatable many times so its relocation table is not overwritten after first use.
- Relocatable Modules are loaded under the control of a Relocatable Module Area (RMA) manager which decides where to load a module initially and where to move each module to whenever the RMA is reorganised. In contrast, an application is loaded at its load address and is then on its own until it exits or faults. An application can only be moved by itself (and then only once, before it begins execution proper).

Self-relocation code for relocatable modules

In this case there is no AIF header, the code must be executable many times, and it must be symbolically addressable from the Relocatable Module header. The code below must be the last area of the RMF image, following the relocation list. Note that it is best thought of as an additional area.

When the following code is executed, the module image has already been loaded at/moved to its target address. It only remains to relocate location-dependent addresses. The list of offsets to be relocated, terminated by (-1), immediately follows End. Note that the address values here (eg `__RelocCode1`) will appear in the list of places to be relocated, allowing the code to be re-executed.

```

IMPORT  Image$$RO$$Base1      ; where the image is linked at...
EXPORT  __RelocCode1         ; referenced from the RM header

__RelocCode1
LDR     R1,      RelocCode1    ; value of __RelocCode (before relocation)
SUB     R11,    PC,      #12   ; value of __RelocCode now
SUBS   R1,     R11,    R1     ; relocation offset
MOVEQS PC,     LR             ; relocate by 0 so nothing to do
LDR     R11,    ImageBase     ; image base prior to relocation...
ADD     R11,    R11,    R1     ; ...where the image really is
ADR     R2,     End

RelocLoop
LDR     R0,     [R2],    #4    ;
CMNS   R0,     #1            ; got list terminator?
MOVLES PC,     LR           ; yes => return
LDR     R3,    [R11, R0]     ; word to relocate
ADD     R3,    R3,    R1     ; relocate it
STR     R3,    [R11, R0]     ; store it back
B       RelocLoop          ; and do the next one

RelocCode1 DCD     __RelocCode1
ImageBase DCD     Image$$RO$$Base1
End        ; the list of locations to relocate
           ; starts here (each is an offset from the
           ; base of the module) and is terminated
           ; by -1.

```

Note that this code, and the associated list of locations to relocate, is added automatically to a relocatable module image by Link (as a consequence of using Link with the Setup option Module enabled).

Self-move and self-relocation code for AIF

This code is added to the end of an AIF image by Link, immediately before the list of relocations (terminated by -1). Note that the code is entered via a BL from the second word of the AIF header so, on entry, R14 points to AIFHeader + 8.

```

RelocCode ROUT
    BIC    R11,    LR,    #&FC000003 ; clear flag bits; -> AIF header + &08
    SUB    R11,    R11,    #8 ; -> header address
    MOV    R0,    #&FB000000 ; BLNV #0
    STR    R0,    [R11, #4] ; won't be called again on image re-entry
;does the code need to be moved?
    LDR    R9,    [R11, #&2C] ; min free space requirement
    CMPS   R9,    #0 ; 0 => no move, just relocate
    BEQ    RelocateOnly
;calculate the amount to move by...
    LDR    R0,    [R11, #&20] ; image zero-init size
    ADD    R9,    R9,    R0 ; space to leave = min free + zero init
    SWI    GetEnv ; MemLimit -> R1
    ADR    R2,    End ; -> End
01 LDR    R0,    [R2], #4 ; load relocation offset, increment R2
    CMNS   R0,    #1 ; terminator?
    BNE    %B01 ; No, so loop again
    SUB    R3,    R1,    R9 ; MemLimit - freeSpace
    SUBS   R0,    R3,    R2 ; amount to move by
    BLE    RelocateOnly ; not enough space to move...
    BIC    R0,    R0,    #15 ; a multiple of 16...
    ADD    R3,    R2,    R0 ; End + shift
    ADR    R8,    %F01 ; intermediate limit for copy-up
;
; copy everything up memory, in descending address order, branching
; to the copied copy loop as soon as it has been copied.
;
02 LDMDB R2!,    {R4-R7}
    STMDB R3!,    {R4-R7}
    CMP    R2,    R8 ; copied the copy loop?
    BGT    %B02 ; not yet
    ADD    R4,    PC,    R0
    MOV    PC,    R4 ; jump to copied copy code
03 LDMDB R2!,    {R4-R7}
    STMDB R3!,    {R4-R7}
    CMP    R2,    R11 ; copied everything?
    BGT    %B03 ; not yet
    ADD    R11,    R11,    R0 ; load address of code
    ADD    LR,    LR,    R0 ; relocated return address
RelocateOnly
    LDR    R1,    [R11, #&28] ; header + &28 = code base set by Link
    SUBS   R1,    R11,    R1 ; relocation offset
    MOVEQ  PC,    LR ; relocate by 0 so nothing to do
    STR    R11,    [R11, #&28] ; new image base = actual load address
    ADR    R2,    End ; start of reloc list
RelocLoop
    LDR    R0,    R2], #4 ; offset of word to relocate
    CMNS   R0,    #1 ; terminator?
    MOVEQS PC,    LR ; yes => return
    LDR    R3,    [R11, R0] ; word to relocate
    ADD    R3,    R3,    R1 ; relocate it
    STR    R3,    [R11, R0] ; store it back
    B     RelocLoop ; and do the next one
    End ; The list of offsets of locations to relocate
; starts here; terminated by -1.

```


ASD

Acknowledgement: This design is based on work originally done for Acorn Computers by Topexpress Ltd.

This section describes the format of symbolic debugging data generated by ARM compilers and assemblers running under RISC OS and used by the desktop debugger DDT.

For each separate compilation unit (called a *section*) the compiler produces debugging data in a special AREA of the object code (see the section entitled *AOF* on page 4-421 for an explanation of AREAs and their attributes). Debugging data are position independent, containing only relative references to other debugging data within the same section and relocatable references to other compiler-generated AREAs.

Debugging data AREAs are combined by the linker into a single contiguous section of a program image (see the section entitled *AIF* on page 4-438 for a description of Application Image Format). Because the debugging section is position-independent, the debugger can move it to a safe location before the image starts executing. If the image is not executed under debugger control the debugging data is simply overwritten.

The format of debugging data allows for a variable amount of detail. This potentially allows the user to trade off among memory used, disc space used, execution time, and debugging detail.

Assembly-language level debugging is also supported, though in this case the debugging tables are generated by the linker, not by language processors. These low-level debugging tables appear in an extra section item, as if generated by an independent compilation. Low-level and high-level debugging are orthogonal facilities, though DDT allows the user to move smoothly between levels if both sets of debugging data are present in an image.

Order of Debugging Data

A debug data AREA consists of a series of *items*. The arrangement of these items mimics the structure of the high-level language program itself.

For each debug AREA, the first item is a section item, giving global information about the compilation, including a code identifying the language and flags indicating the amount of detail included in the debugging tables.

Each data, function, procedure, etc., definition in the source program has a corresponding debug data item and these items appear in an order corresponding to the order of definitions in the source. This means that any nested structure in the source program is preserved in the debugging data and the debugger can use this structure to make deductions about the scope of various source-level objects. Of course, for

procedure definitions, two debug items are needed: a **procedure** item to mark the definition itself and an **endproc** item to mark the end of the procedure's body and the end of any nested definitions. If procedure definitions are nested then the procedure - endproc brackets are also nested. Variable and type definitions made at the outermost level, of course, appear outside of all procedure/endproc items.

Information about the relationship between the executable code and source files is collected together and appears as a **fileinfo** item, which is always the final item in a debugging AREA. Because of the C language's #include facility, the executable code produced from an outer-level source file may be separated into disjoint pieces interspersed with that produced from the included files. Therefore, source files are considered to be collections of 'fragments', each corresponding to a contiguous area of executable code and the fileinfo item is a list with an entry for each file, each in turn containing a list with an entry for each fragment. The fileinfo field in the section item addresses the fileinfo item itself. In each procedure item there is a 'file entry' field which refers to the file-list entry for the source file containing the procedure's start; there is a separate one in the endproc item because it may possibly not be in the same source file.

Representation of Data Types

Several of the debugging data items (eg procedure and variable) have a **type** word field to identify their data type. This field contains, in the most significant 3 bytes, a code to identify a base type and, in the least significant byte, a pointer count: 0 to denote the type itself; 1 to denote a pointer to the type; 2 to denote a pointer to a pointer to...; etc.

For simple types the code is a positive integer as follows:

void	0	(all codes are decimal)
signed integers		
single byte	10	
half-word	11	
word	12	
unsigned integers		
single byte	20	
half-word	21	
word	22	
floating point		
float	30	
double	31	
long double	32	

complex	
single complex	41
double complex	42
functions	
function	100

For compound types (arrays, structures, etc.) there is a special kind of debug data item (**array**, **struct**, etc.) to give details of the type such as array bounds and field types. The type code for such types is negative being the negation of the (byte) offset of the special item from the start of the debugging AREA.

If a type has been given a name in a source program, it will give rise to a **type** debugging data item which contains the name and a type word as defined above. If necessary, there will also be a debugging data item such as an array or struct to define the type itself. In that case, the type word will refer to this item.

Enumerated types in C and scalars in Pascal are treated simply as integer sub-ranges of an appropriate size, the name information is not available in this version of the debugging format. Set types in Pascal are not treated in detail: the only information recorded for them is the total size occupied by the object in bytes.

Fortran character types are supported by a special kind of debugging data item the format of which is yet to be defined.

Representation of Source File Positions

Several of the debugging data items have a **sourcepos** field to identify a position in the source file. This field contains a line number and character position within the line packed into a single word. The most significant 10 bits encode the character offset (0-based) from the start of the line and the least- significant 22 bits give the line number.

Debugging Data Items in Detail

The first word of each debugging data item contains the byte length of the item (encoded in the most significant 16 bits) and a code identifying the kind of item (in the least significant 16 bits). The following codes are defined:

1	section
2	procedure
3	endproc
4	variable
5	type
6	struct
7	array

8	subrange
9	set
10	fileinfo

The meaning of the second and subsequent words of each item is defined below.

Where items include a string field, the string is packed into successive bytes beginning with a length byte, and padded at the end to a word boundary (the padding value is immaterial, but NUL or ‘ ’ is preferred). The length of a string is in the range [0 - 255] bytes.

Where an item contains a field giving an offset in the debugging data area (usually to address another item), this means a byte offset from the start of the debugging data for the whole section (in other words, from the start of the section item).

Section

A section item is the first item of each section of the debugging data. The first five fields are held in a single word:

language	one byte code identifying the source language
debuglines	1 bit: set \Rightarrow tables contain line numbers
debugvars	1 bit: set \Rightarrow tables contain data about local variables
spare	14 reserved bits (must be zero)
debugversion	one byte version number of the debugging data
codeaddr	pointer to start of executable code in this section
dataaddr	pointer to start of static data for this section
codesize	byte size of executable code in this section
datasize	byte size of the static data in this section
fileinfo	offset in the debugging data of the file information for this section (or 0 if no fileinfo is present)
debugsize	total byte length of debugging data for this section
name or nsyms	string or integer

The name field contains the program name for Pascal and Fortran programs. For C programs it contains a name derived by the compiler from the main file name (notionally a module name). Its syntax is similar to that for a variable name in the source language. For a low-level debugging section (language = 0) the field is treated as a 4 byte integer giving the number of symbols following.

The following language byte codes are defined:

0	Low-level debugging data (notionally, assembler)
1	C
2	Pascal
3	Fortran77
other	reserved to Acorn.

The fileinfo field is 0 if no source file information is present.

The debugversion field was defined to be 1; the new debugversion for the extended debugging data format (encompassing low-level debugging data) is 2. For low-level debugging data, other fields have the following values:

language	0
codeaddr	Image\$\$RO\$\$Base
dataaddr	Image\$\$RW\$\$Base
codesize	Image\$\$RO\$\$Limit - Image\$\$RO\$\$Base
datasize	Image\$\$RW\$\$Limit - Image\$\$RW\$\$Base
fileinfo	0
nsyms	number of symbols within the following debugging data
debugsize	total size of the low-level debugging data including the size of the section item

The section item is immediately followed by nsyms symbols, each having the following format:

stridx:24	byte offset in string table of symbol name
flags:8	(see below)
value	the value of the symbol

The flags field has the following values:

0/1	the symbol is a local/global symbol
+	(there may be many local symbols with the same name)
0/2/4/6	symbol names an absolute/code/data/zero-init value

Note that the linker reduces all symbol values to absolute values. The flags field records the history, or origin, of the symbol in the image.

The string table is in standard AOF format. It consists of a length word followed by the strings themselves, each terminated by a NUL (0). The length word includes the length of the length word, so no offset into the string table is less than 4. The end of the string table is padded to the next word boundary.

Procedure

A procedure item appears once for each procedure or function definition in the source program. Any definitions with the procedure have their related debugging data items between the procedure item and the matching endproc item. The format of procedure items is as follows:

type	the return type if this is a function, else 0
args	the number of arguments
sourcepos	a word encoding the source position of the start of the procedure

startaddr	pointer to the first instruction of the procedure
bodyaddr	pointer to the first instruction of the procedure body (see below)
endproc	offset of the related endproc item
fileentry	offset of the file list entry for the source file
name	string

The bodyaddr field points to the first instruction after the procedure entry sequence, that is the first address at which a high-level breakpoint could sensibly be set. The startaddr field points to the beginning of the entry sequence, that is the address at which control actually arrives when the procedure is called.

A label in a source program is represented by a special procedure item with no matching endproc (the endproc field is 0 to denote this). Pascal and Fortran numerical labels are converted by the compiler into strings prefixed by '\$n'.

For Fortran77, multiple entry points to the same procedure each give rise to a separate procedure item but they all have the same endproc offset referring to a single endproc item.

Endproc

This item marks the end of the debugging data items belonging to a particular procedure. It also contains information relating to the procedure's return. Its format is as follows:

sourcepos	a word encoding the position in the source file of the end of the procedure
endaddr	a pointer to the code byte AFTER the compiled code for the procedure
fileentry	offset of the file list entry for the procedure's end
nreturns	number of procedure return points (may be 0)
retadds...	pointers to the procedure-return code

If the procedure body is an infinite loop, there will be no return point so nreturns will be 0. Otherwise the retadds should each point to a suitable location at which a breakpoint may be set 'at the exit of the procedure'. When execution reaches this point, the current stack frame should still be in this procedure.

Variable

This item contains debugging data relating to a source program variable or a formal argument to a procedure (the first variable items in a procedure always describe its arguments). Its format is as follows:

type	a type word
sourcepos	a word encoding the source position of the variable
class	a word encoding the variable's storage class
location	see explanation below
name	string

The following codes define the storage classes of variables:

1	external variables (or Fortran common)
2	static variables private to one section
3	automatic variables
4	register variables
5	Pascal var arguments
6	Fortran arguments
7	Fortran character arguments

The meaning of the location field of a variable item depends on the storage class: it contains an absolute address for static and external variables (relocated by the linker); a stack offset (ie an offset from the frame- pointer) for automatic and var-type arguments; an offset into the argument list for Fortran arguments; and a register number for register variables (the 8 floating point registers are numbered 16 - 23).

No account is taken of variables which ought to be addressed by +ve offsets from the stack-pointer rather than -ve offsets from the frame-pointer.

The sourcepos field is used by the debugger to distinguish between different definitions having the same name (eg identically named variables in disjoint source-level naming scopes such as nested block in C).

Type

This item is used to describe a named type in the source language (eg a typedef in C). The format is as follows:

type	a type word (described earlier)
name	string

Struct

This item is used to describe a structured data type (eg a struct in C or a record in Pascal). Its format is as follows:

fields	the number of fields in the structure
size	total byte size of the structure
fieldtable...	a table of fields entries in the following format:

offset	byte offset of this field within the structure
type	a type word (interpretation as described earlier)
name	string

Union types are described by struct items in which all fields have 0 offsets.

C bit fields are not treated in full detail: a bit field is simply represented by an integer starting on the appropriate word boundary (so that the word contains the whole field).

Array

This item is used to describe a one-dimensional array. Multi-dimensional arrays are described as arrays of arrays. Which dimension comes first is dependent on the source language (different for C and Fortran). The format is as follows:

size	total byte size of each element
arrayflags	(see below)
basetype	a type word
lowerbound	constant value or stack offset of variable
upperbound	constant value or stack offset of variable

If the size field is zero, debugger operations affecting the whole array, rather than individual elements of it, are forbidden.

The following bit numbers in the arrayflags field are defined:

0	lower bound is undefined
1	lower bound is a constant
2	upper bound is undefined
3	upper bound is a constant

If a bound is defined and not constant then it is an integer variable on the stack and the boundvalue field contains the stack offset of the variable (from the frame-pointer).

Subrange

This item is used to describe subrange typed in Pascal. It also serves to describe enumerated types in C and scalars in Pascal (in which case the base type is understood to be an unsigned integer of appropriate size). Its format is as follows:

size	half-word: 1, 2, or 4 to indicate byte size of object
typecode	half-word: simple type code
lwb	lower bound of subrange
upb	upper bound of subrange

Set

This item is used to describe a Pascal set type. Currently, the description is only partial. The format is:

size byte size of the object

Fileinfo

This item appears once per section after all other debugging data items. The half of the header word which would usually give the item length is not required and should be set to 0.

Each source file is described by a sequence of 'fragments', each of which describes a contiguous region of the file within which the addresses of compiled code increase monotonically with source-file position. The order in which fragments appear in the sequence is not necessarily related to the source file positions to which they refer.

Note that for compilations that make no use of the #include facility, the list of fragments will have only one entry and all line-number information will be contiguous.

The item is a list of entries each with the following format:

length	length of this entry in bytes (0 marks the final entry)
date	date and time when the file was last modified
filename	string (or null if the name is not known)
n	number of fragments following
fragments...	n fragments with the following structure...
fragmentsize	length of this entry in bytes
firstline	linenumber
lastline	linenumber
codeaddr	pointer to the start of the fragment's executable code
codesize	byte size of the code in the fragment
lineinfo...	a variable number of line number data

There is one lineinfo half-word for each statement of the source file fragment which gives rise to executable code. Exactly what constitutes an executable statement may be defined by the language implementation; the definition may for instance include some declarations. The half-word can be regarded as 2 bytes: the first contains the number of bytes of code generated from the statement and cannot be zero; the second contains the number of source lines occupied by the statement (ie the difference between the line number of the start of the statement and the line number of the next statement). This may be zero if there are multiple statements on the same source line.

If the whole half-word is zero, this indicates that one of the quantities is too large to fit into a byte and that the following 2 half-words contain (in order) the number of lines followed by the number of bytes of code generated from the statement.



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, Outlines and bitmap files
- Music files
- Squash files

Sprite files

A sprite file is saved in the same format as a sprite area is saved in memory, except that the first word of the sprite area is not saved.

For a full description of sprite area formats, refer to the section entitled *Format of a sprite area* on page 1-779.

Template files

The following section describes the Wimp template file format:

Header

The file starts with a header:

Size	Description
4	file offset of font data (-1 if none)
4	reserved (must be zero)
4	reserved (must be zero)
4	reserved (must be zero)

Index entries

The header is followed by a series of index entries to data later in the file:

Size	Description
4	file offset of data for this entry
4	size of data for this entry
4	entry type (1 = window)
12	identifier (control character terminated)

Terminator

The index entries are terminated by a null word:

Size	Description
4	0

Data

Each set of entries referred to earlier in the index contains the following:

Size	Description
88	window definition (as in <code>Wimp_CreateWindow</code> – see page 3-87)
$ni \times 32$	icon definitions (as in <code>Wimp_CreateIcon</code> – see page 3-93)
?	indirected icon data

Any pointers to indirected icon data are offsets from the start of the current entry. Any references to anti-aliased fonts use internal handles.

Font data

The file ends with an optional set of font data (the presence of which is indicated by the first word of the header):

Size	Description
4	x-point-size × 16
4	y-point-size × 16
40	font name (control character terminated)

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 contact Acorn; see *Appendix H: Registering names* on page 4-549.

Coordinates

All coordinates within a Draw file are signed 32-bit integers that give absolute positions on a large image plane. The units are $1/(180 \times 256)$ 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 0, 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 characters, padded with spaces
4	x-low
4	y-low
4	x-high
4	y-high

} bounding box
} bottom-left (x-low, y-low) is inclusive
} top-right (x-high, y-high) is exclusive

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 'x-low' value may be greater than the 'x-high' value (and similarly for the y values).

Objects

Each object consists of an object header, followed by a variable amount of data depending on the object type.

Object header

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
4	x-low
4	y-low
4	x-high
4	y-high

} object bounding box
bottom-left (x-low, y-low) is inclusive
top-right (x-high, y-high) is exclusive

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 (hence the header is only two words long). Such objects 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

A font table object has no bounding box in its object header, which is followed by a sequence of font number definitions:

Size	Description
1	font number (non-zero)
<i>n</i>	<i>n</i> character textual font name, null terminated
0 - 3	up to 3 zero characters, to pad to a 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. 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	unsigned nominal x size of the font (in 1/640 point)
4	unsigned nominal y size of the font (in 1/640 point)
8	x, y coordinates of the start of the text base line
<i>n</i>	<i>n</i> characters in the string, null terminated
0 - 3	up to 3 zero characters, to pad to a word boundary

The character string consists of printing ANSI characters with codes within the ranges 32 - 126 and 128 - 255. This need not be checked during rendering, but other codes may produce undefined or system-dependent results.

The text style word consists of the following:

Bit(s)	Description
0 - 7	one byte font number
8 - 31	reserved (must be zero)

Italic, bold variants etc 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 nominal x 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 \Rightarrow do not fill)
4	outline colour (-1 \Rightarrow 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:

Size	Description
4	1-word <i>tag identifier</i> : bits 0 - 7 = tag identifier byte: 0 \Rightarrow end of path: no arguments 2 \Rightarrow move to absolute position: followed by one x, y pair 5 \Rightarrow close current sub-path: no arguments 8 \Rightarrow draw to absolute position: followed by one x, y pair 6 \Rightarrow Bezier curve through two control points, to absolute position: followed by three x, y pairs bits 8 - 31 reserved (must be zero)
$n \times 8$	sequence of n 2-word (x, y) coordinate pairs (where n is zero, one or three, as determined by the value of the <i>tag identifier</i>)

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 chapter entitled *Draw module* on page 3-533 for precise rules concerning the appearance of paths. See also manuals on PostScript. (Reference: *PostScript Language Reference Manual*. Adobe Systems Incorporated (1990) 2nd ed. Addison-Wesley, Reading, Mass, USA).

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/or '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 as end cap style)
6	winding rule: 0 = non-zero 1 = even-odd
7	dash pattern: 0 = dash pattern missing 1 = dash pattern present
8 - 15	reserved (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

followed by, for each element of the dash pattern:

Size	Description
4	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. See the section entitled *Format of a sprite* on page 1-779 for details.

This contains a pixelmap 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 – for further details see the chapter entitled *Sprites* on page 1-775 and the chapter entitled *The Window Manager* on page 3-3.

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 contact Acorn; see *Appendix H: Registering names* on page 4-549.

Text area object

Object type number 9

Size	Description
?	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 (must be zero)
4	reserved (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	up to 3 zero characters, to pad 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
● <code>\! <version><newline or /></code>	Must appear at the start of the text, and only there. <code><version></code> must be 1.
● <code>\A<code><optional /></code>	Set alignment. <code><code></code> is one of L (left = default), R (right), C (centre), D (double). A change in alignment forces a line break.
● <code>\B<R><spaces><G><spaces><newline or /></code>	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...+127, and <thickness> to 0...255. To turn the underlining off, either give a thickness of 0, or use the command `'\U.'`
- `\W[-]<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 owing 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.

Options object

Object type number 11

The object header for an options object has space allocated for a bounding box, but since one would be meaningless, the space is unused. You must treat the 4 words normally used for the bounding box as reserved, and set them to zero.

Size	Description
4	(paper size + 1) × &100 (ie &500 for A4)
4	paper limits options: bit 0 set ⇒ paper limits shown bits 1 - 3 reserved (must be zero) bit 4 set ⇒ landscape orientation (else portrait) bits 5 - 7 reserved (must be zero) bit 8 set ⇒ printer limits are default bits 9 - 31 reserved (must be zero)
8	grid spacing (floating point)
4	grid division
4	grid type (zero ⇒ rectangular, non-zero ⇒ isometric)
4	grid auto-adjustment (zero ⇒ off, non-zero ⇒ on)
4	grid shown (zero ⇒ no, non-zero ⇒ yes)
4	grid locking (zero ⇒ off, non-zero ⇒ on)
4	grid units (zero ⇒ inches, non-zero ⇒ centimetres)
4	zoom multiplier (1 - 8)
4	zoom divider (1 - 8)
4	zoom locking (zero ⇒ none, non-zero ⇒ locked to powers of two)
4	toolbox presence (zero ⇒ no, non-zero ⇒ yes)
4	initial entry mode: one of bit 0 set ⇒ line bit 1 set ⇒ closed line bit 2 set ⇒ curve bit 3 set ⇒ closed curve bit 4 set ⇒ rectangle bit 5 set ⇒ ellipse bit 6 set ⇒ text line bit 7 set ⇒ select
4	undo buffer size, in bytes

When Draw reads a draw file, only the first options object is taken into account – any others are discarded. When it saves a diagram to file, the options in force for that diagram are saved with it.

The Draw application supplied with RISC OS 2 does not use this object type.

Transformed text object

Object type number 12

Size	Description
24	transformation matrix
4	font flags: bit 0 set \Rightarrow text should be kerned bit 1 set \Rightarrow text written from right to left bits 2 - 31 reserved (must be zero)
4	text colour
4	text background colour hint
4	text style
4	unsigned nominal x size of the font (in 1/640 point)
4	unsigned nominal y size of the font (in 1/640 point)
8	x, y coordinates of the start of the text base line
<i>n</i>	<i>n</i> characters in the string, null terminated
0 - 3	up to 3 zero characters, to pad to a word boundary

The transformation matrix is as described in Font_Paint (see page 3-437), in the same format used elsewhere in the Draw module.

The remaining fields are exactly as specified for Text objects (see page 4-464).

The Draw application supplied with RISC OS 2 does not use this object type.

Transformed sprite object

Object type number 13

Size	Description
24	Transformation matrix

This is followed by a sprite. See the section entitled *Format of a sprite* on page 1-779 for details.

This contains a pixelmap image. The image is transformed from its own coordinates (ie the bottom-left at (0, 0) and the top-right at $w \times x_eig, h \times y_eig$), where (w, h) are the width and height of the sprite in pixels, and (x_eig, y_eig) are the eigen factors for the mode in which it was defined) by the transformation held in the matrix.

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 – for further details see the chapter entitled *Sprites* on page 1-775 and the chapter entitled *The Window Manager* on page 3-3.

The Draw application supplied with RISC OS 2 does not use this object type.

Font files

In all the formats described below, 2-byte quantities are little-endian: that is, the least significant byte comes first, followed by the most-significant. The values are unsigned unless otherwise stated.

Fonts are described in:

- IntMetrics and IntMetn files
- x90y45 files (old style 4-bpp bitmaps)
- New font file formats.

IntMetrics / IntMetn files

Header

Size	Description
40	name of font, padded with Return characters
4	16
4	16
1	<i>nlo</i> = low byte of number of characters that may be defined
1	<i>version number</i> of file format: <ul style="list-style-type: none"> 0 <i>flags</i> and <i>nhi</i> must be zero 1 not supported 2 <i>flags</i> supported; <i>n</i> can be > 255
1	<i>flags</i> : <ul style="list-style-type: none"> bit 0 set ⇒ there is no bbox data (use Outlines) bit 1 set ⇒ there is no x-offset data bit 2 set ⇒ there is no y-offset data bit 3 set ⇒ there is more data after the metrics bit 4 reserved (must be zero) bit 5 set ⇒ character map size precedes map bit 6 set ⇒ kern characters are 16-bit, else 8-bit bit 7 reserved (must be zero)
1	<i>nhi</i> = high byte of number of characters that may be defined: $n = nhi \times 256 + nlo$
If <i>flags</i> bit 5 is set:	
2	<i>m</i> = character map size 0 ⇒ no map

Some of the n character definitions can be blank; the number defines the number of slots available – though not necessarily used – in the character definition tables.

Character mapping

Size	Description
m	character mapping (ie indices into following tables) For example, if the 40th byte in this mapping has the value 4, then the fourth entry in each of the following arrays refers to character 40. A zero entry means that character is not defined in this font. If <i>flags</i> bit 5 is clear, 256 characters are mapped (ie $m = 256$).

If there is no map (see above), the character code is used to index into the tables.

Note that since the mapping table is 8-bit, there cannot be one if $n > 256$.

Table of bounding boxes

If *flags* bit 0 is clear:

Size	Description
$2n$	x0
$2n$	y0
$2n$	x1
$2n$	y1

} bounding box for each character (16-bit signed)
 } bottom-left (x0, y0) is inclusive
 } top-right (x1, y1) is exclusive
 } coordinates are in 1/1000th em

Coordinates are relative to the ‘origin point’.

Tables of character widths

If *flags* bit 1 is clear:

Size	Description
$2n$	x-offset after printing each character, in 1/1000th em (16-bit signed)

If *flags* bit 2 is clear:

Size	Description
$2n$	y-offset after printing each character, in 1/1000th em (16-bit signed)

To calculate the offset to this point in the file, let:

nlo = byte at offset 48 in file
 $version\ number$ = byte at offset 49 in file
 $flags$ = byte at offset 50 in file
 nhi = byte at offset 51 in file
 If $version\ number < 2$ then $flags = 0$ (which it should be anyway!)
 $n = nhi \times 256 + nlo$

Then:

offset = 52
 if ($flags$ bit 5 clear) then offset += 256
 else offset += 2 + byte(52) + 256 × byte(53)
 if ($flags$ bit 0 clear) then offset += $8n$
 if ($flags$ bit 1 clear) then offset += $2n$
 if ($flags$ bit 2 clear) then offset += $2n$

Offsets to extra data areas

If $flags$ bit 3 is set:

Size	Description
2	offset to ‘miscellaneous’ data area
2	offset to kern pair data area
2	offset to reserved data area #1
2	offset to reserved data area #2

The offsets are relative to the start of the table. The entries must be consecutive in the file, so the end of one area coincides with the beginning of the next. The areas are not necessarily word-aligned, and the space at the end of each area is reserved (ie there must not be any ‘dead’ space at the end of an area).

Miscellaneous data area

Size	Description
2	x0
2	y0
2	x1
2	y1
2	default x-offset per char (if <i>flags</i> bit 1 is set), in 1/1000th em (16-bit signed)
2	default y-offset per char (if <i>flags</i> bit 2 is set), in 1/1000th em (16-bit signed)
2	italic h-offset per em ($-1000 \times \text{TAN}$ (italic angle)) (16-bit signed)
1	underline position, in 1/256th em (signed)
1	underline thickness, in 1/256th em (unsigned)
2	CapHeight in 1/1000th em (16-bit signed)
2	XHeight in 1/1000th em (16-bit signed)
2	Descender in 1/1000th em (16-bit signed)
2	Ascender in 1/1000th em (16-bit signed)
4	reserved (must be zero)

Kern pair data

If *flags* bit 6 is clear, character codes are 8-bit; if *flags* bit 6 is set, character codes are 16-bit (lo, hi).

Size	Description
1 or 2	left-hand character code
1 or 2	right-hand character code
2	x-kern amount (if <i>flags</i> bit 1 is clear) in 1/1000ths em (16-bit signed)
2	y-kern amount (if <i>flags</i> bit 2 is clear) in 1/1000ths em (16-bit signed)
1 or 2	0 \Rightarrow end of list for this letter
1 or 2	0 \Rightarrow end of kern pair data

Reserved data areas #1 and #2

These must be null.

x90y45 font files

If the length of a x90y45 file is less than 256 bytes, then the contents are the name of the f9999x9999 file to use as master bit maps.

Index entries

Each font file starts with a series of 4-word (ie 16 byte) 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 in the x-direction
1	pixels per inch in the y-direction
4	reserved (must be zero)
4	offset of pixel data in file
4	size of pixel data

The list is terminated by:

1	0
---	---

Pixel data

Pixel data is limited to 64KBytes per block. Each block starts word-aligned relative to the start of the file:

Size	Description
4	x-size, in 1/16ths point \times x pixels per inch
4	y-size, in 1/16ths point \times y pixels per inch
4	pixels per inch in the x-direction
4	pixels per inch in the y-direction
1	x0
1	y0
1	x1
1	y1
512	2-byte offsets from table start of character data.

maximum bounding box for font
 bottom-left (x0, y0) is inclusive
 top-right (x1, y1) is exclusive
 all coordinates are in pixels

A zero value means the character is not defined.

Character data

Size	Description
1	x0
1	y0
1	$x1 - x0 = X$
1	$y1 - y0 = Y$
$X \times Y / 2$	4-bits per pixel (bpp), consecutive rows bottom to top: not aligned until the end
0 - 3.5	alignment

} bounding box for character
 bottom-left (x0, y0) is inclusive
 top-right (x1, y1) is exclusive
 all coordinates are in pixels

Other 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 $\times 16 \times \text{dpi} / 72$) zero-suppressed decimal number.

If the length of an outlines file is less than 256 bytes, then the contents are the name of another font whose glyphs are to be used instead (with this font’s metrics).

File header

The file header is of the following form:

Size	Description								
4	'FONT' – identification word								
1	<i>bpp</i> (bits per pixel): 0 ⇒ outlines 1 ⇒ 1 bpp 4 ⇒ 4 bpp								
1	<i>version number</i> of file format (changes are cumulative): 4 no 'don't draw skeleton lines unless smaller than this' byte present 5 byte at [table+512] = maximum pixel size for skeleton lines (see below) 6 byte at [chunk + indexsize] = dependency mask (see below) 7 flag word precedes index in chunk (offsets are relative to index, not chunk) 8 file offset array is in a different place								
2	If <i>bpp</i> = 0: design size of font If <i>bpp</i> > 0: flags: bit 0 set ⇒ horizontal subpixel placement bit 1 set ⇒ vertical subpixel placement bits 2-5 reserved (must be zero) bit 6 set ⇒ flag word precedes index in chunk (must be set if <i>version number</i> ≥ 7, else clear). bit 7 reserved (must be zero) Outline files derive the value of bit 6 from <i>version number</i> .								
2	<table border="0"> <tr> <td>x0</td> <td rowspan="4">} maximum bounding box for font (16-bit signed)</td> </tr> <tr> <td>y0</td> <td>bottom-left (x0, y0) is inclusive</td> </tr> <tr> <td>x1 – x0</td> <td>top-right (x1, y1) is exclusive</td> </tr> <tr> <td>y1 – y0</td> <td>all coordinates are in pixels or design units</td> </tr> </table>	x0	} maximum bounding box for font (16-bit signed)	y0	bottom-left (x0, y0) is inclusive	x1 – x0	top-right (x1, y1) is exclusive	y1 – y0	all coordinates are in pixels or design units
x0		} maximum bounding box for font (16-bit signed)							
y0				bottom-left (x0, y0) is inclusive					
x1 – x0				top-right (x1, y1) is exclusive					
y1 – y0	all coordinates are in pixels or design units								
2									
2									
2									

If *version number* < 8, the number of chunks *nchunks* = 8, and these bytes end the header:

Size	Description
4	file offset of 0...31 chunk (word-aligned)
4	file offset of 32...63 chunk (word-aligned)
20	file offsets of further chunks, in order (word-aligned)
4	file offset of 224...255 chunk (word-aligned)

4 file offset of end (ie size of file)
 If $\text{offset}(n+1) = \text{offset}(n)$, then chunk n is null.

If *version number* ≥ 8 , these bytes end the header:

Size	Description
4	file offset of area containing file offsets of chunks
4	<i>nchunks</i> = number of defined chunks
4	<i>ns</i> = number of scaffold index entries (including entry[0] = size)
4	<i>scaffold flags</i> : bit 0 set \Rightarrow all scaffold base chars are 16-bit bit 1 set \Rightarrow these outlines should not be anti-aliased (eg System.Fixed) bits 2 - 31 reserved (must be zero)
4×5	all reserved (must be zero)

Table start

Size	Description
2	n = size of table/scaffold data

Table data

Bitmaps

If *bpp* > 0 , the file defines a bitmap, and only the following 8 bytes of table data are used. For such a file, $n=10$ – other values are reserved.

Size	Description
2	x-size (1/16th point)
2	x-resolution (dpi)
2	y-size (1/16th point)
2	y-resolution (dpi)

Outlines

If $bpp = 0$, the file defines outlines, and the following table data is used. (Files with $version\ number < 8$ behave as if $ns = 256$ and $scaffold\ flags = 0$.)

Size	Description
$ns \times 2 - 2$	offsets to scaffold data (16-bit): If <i>scaffold flags</i> bit 0 is clear: bits 0 - 14 = offset of scaffold data from table start bit 15 set \Rightarrow base character code is 2 bytes, else 1 byte If <i>scaffold flags</i> bit 0 is set: bits 0 - 15 = offset of scaffold data from table start base character code is always 2 bytes 0 \Rightarrow no scaffold data for char
1	skeleton threshold pixel size (if $version\ number \geq 5$) When rastering the outlines, skeleton lines will only be drawn if either the x- or the y- pixel size is less than this value (except if value = 0, which means 'always draw skeleton lines').
?	... sets of scaffold data follow, each set of which can include many scaffold lines (see descriptions below)

Scaffold data

Size	Description
1	character code of 'base' scaffold entry (0 \Rightarrow none)
1	bit n set \Rightarrow x-scaffold line n is defined in base character
1	bit n set \Rightarrow y-scaffold line n is defined in base character
1	bit n set \Rightarrow x-scaffold line n is defined locally
1	bit n set \Rightarrow y-scaffold line n is defined locally ... local scaffold lines follow (see description below)

Scaffold lines

Size	Description
2	bits 0 - 11 = coordinate in 1/1000ths em (signed) bits 12 - 14 = scaffold link index (0 \Rightarrow none) bit 15 set \Rightarrow 'linear' scaffold line
1	width (254 \Rightarrow L-tangent, 255 \Rightarrow R-tangent)

Table end

Size	Description
?	description of contents of file: <i>Font name</i> , 0, 'Outlines', 0, '999x999 point at 999x999 dpi', 0 ... word-aligned chunk data follows (see description below)

If *version number* ≥ 8 :

Size	Description
4	file offset of chunk 0 (word-aligned)
4	file offset of chunk 1 (word-aligned)
$4 \times (nchunks-3)$	file offset of further chunks in order (word-aligned)
4	file offset of chunk ($nchunks - 1$) (word-aligned)
4	file offset of end (ie size of file)

Chunk data

If *version number* ≥ 7 :

Size	Description
4	flag word: bit 0 set \Rightarrow horizontal subpixel placement bit 1 set \Rightarrow vertical subpixel placement bits 2 - 6 reserved (must be zero) bit 7 set \Rightarrow dependency byte(s) present (see below) bits 8 - 30 reserved (must be zero) bit 31 reserved (must be one)

For all versions there follow *nchunks* of chunk data in this format:

Size	Description
32	offset within chunk to character data 0 ⇒ character is not defined Size is × 4 if vertical placement is used, and × 4 if horizontal placement is used (because the character data is repeated for each of four possible sub-placements). Character index is more tightly bound than vertical placement, which is more tightly bound than horizontal placement.
?	dependency bytes (if outline file, and <i>version number</i> ≥ 6) One bit required for each chunk in file. Bit <i>n</i> set ⇒ chunk <i>n</i> must be loaded in order to rasterise this chunk. This is required for composite characters which include characters from other chunks (see below). ...character data follows, word-aligned at end of chunk (see description 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.

Character data

Size	Description
1	<i>character flags</i> : bit 0 set ⇒ coordinates 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 If <i>character flags</i> bit 3 is clear: bits 4 - 7 = 'f' value for char (0 ⇒ not encoded) If <i>character flags</i> bit 3 is set: bit 4 set ⇒ composite character bit 5 set ⇒ with an accent as well bit 6 set ⇒ character codes within this character are 16-bit, else 8-bit (not yet implemented – must be zero) bit 7 reserved (must be zero)

if *character flags* bits 3 and 4 are set:

Size	Description
1 or 2	character code of base character

if *character flags* bits 3 and 5 are set:

Size	Description
1 or 2	character code of accent
2 or 3	x, y offset of accent character

if *character flags* bits 3 or 4 are clear:

Size	Description
1 or 1.5	x0
1 or 1.5	y0
1 or 1.5	x1 – x0
1 or 1.5	y1 – y0
?	data: (depends on type of file)

} bounding box for character (8- or 12-bit signed)
 } bottom-left (x0, y0) is inclusive
 } top-right (x1, y1) is exclusive
 } all coordinates are in pixels or design units
 1-bpp uncrunched: rows from bottom to top
 4-bpp uncrunched: rows from bottom to top
 1-bpp crunched: list of (packed) run-lengths
 outlines: list of move/line/curve segments

Word-aligned at the end of the character data.

Outline character format

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). For further details see the chapter entitled *Draw module* on page 3-533.

Each line segment consists of:

Size	Description
1	bits 0 - 1 = segment type: <ul style="list-style-type: none"> 0 terminator (see description 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 in design units

Terminator:

Size	Description
1	bit 2 set \Rightarrow stroke paths follow (same format, but paths are not closed) bit 3 set \Rightarrow composite character inclusions follow:

Composite character inclusions:

1 or 2	character code of character to include (0 \Rightarrow 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.

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-bpp uncompact format

1 bit per pixel, bit set \Rightarrow paint in foreground colour, in rows from bottom-left to top-right, not aligned until word-aligned at the end of the character.

1-bpp compacted format

The whole character is initially treated as a stream of bits, as for the uncompact form. The bit stream is then scanned row by row: consecutive duplicate rows are 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* determines 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.

File	Meaning
<i>n</i> nibbles, value 0	run length = $next_n+1_nibbles + (13-f) \times 16 + f+1 - 16$
1 nibble, value 1.. <i>f</i>	run length = <i>this_nibble</i>
1 nibble, value <i>f</i> +1...13	run length = $next_nibble + (this_nibble-f-1) \times 16 + f+1$
1 nibble, value 14	row repeat count = <i>next_packed_number</i>
1 nibble, value 15	row repeat count = 1

where:

- *this_nibble* is the actual value (1..*f*, or *f*+1...13) of the nibble
- *next_nibble* is the actual value (0...15) of the nibble following *this_nibble*
- *next_n+1_nibbles* is the actual value ($0 \dots 2^{4(n+1)} - 1$) of the next *n*+1 nibbles after the *n* zero nibbles
- *next_packed_number* is the value of the packed number following the nibble of value 14.

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.

Encoding files

An encoding file is a text file which contains a set of identifiers which indicate which characters appear in which positions in a font. Each identifier is preceded by a '*'*', and the characters are numbered from 0, increasing by 1 with each identifier found.

Comments are introduced by '*%*', and continue until the next control character.

The following special comment lines are understood by the font manager:

```
%%RISCOS_BasedOn base_encoding
%%RISCOS_Alphabet alphabet
```

where *base_encoding* and *alphabet* denote positive decimal integers.

Both lines are optional, and they indicate respectively the number of the base encoding and the alphabet number of this encoding.

If the %%RISCOS_BasedOn line is not present, then the Font Manager assumes that the target encoding describes the actual positions of the glyphs in an existing file, the data for which is in:

```
font_directory.IntMetricsalphabet
font_directory.Outlinesalphabet
```

where *alphabet* is null if the %%RISCOS_Alphabet line is omitted.

(In fact the leafnames are shortened to fit in 10 characters, by removing characters from just before the *alphabet* suffix).

In this case the IntMetrics and Outlines files are used directly, since there is a one-to-one correspondence between the positions of the characters in the datafiles and the positions required in the font.

If the %%RISCOS_BasedOn line **is** present, then the Font Manager accesses the following datafiles:

```
font_directory.IntMetricsbase_encoding
font_directory.Outlinesbase_encoding
```

and assumes that the positions of the glyphs in the datafiles are as given by the contents of the base encoding file.

The base encoding is called ‘/Basen’, and lives in the Encodings directory under Font\$Path, along with all the other encodings. Because it is preceded by a ‘/’, the Font Manager does not return it in the list of encodings returned by Font_ListFonts.

Note that only one encoding file with a given name can apply to all the fonts known to the system. Because of this, a given encoding can only be either a direct encoding, where the alphabet number is used to reference the datafiles, or an indirect encoding, where the base encoding number specifies the datafile names.

Here is the start of a sample base encoding (‘/Base0’):

```
% /Base0 encoding

%%RISCOS_Alphabet 0

/.notdef /.NotDef /.NotDef /.NotDef
/zero /one /two /three /four /five /six /seven /eight
```

Here is the start of a sample encoding file ('Latin1'):

```
% Latin 1 encoding

%%RISCOS_BasedOn 0
%%RISCOS_Alphabet 101

/.notdef /.notdef /.notdef /.notdef /.notdef /.notdef /.notdef /.notdef
/space /exclam /quotedbl /numbersign
/dollar /percent /ampersand /quotesingle
```

(Note that the sample /Base0 file is not the same as the released one).

These illustrate several points:

- The %% lines must appear before the first identifier.
- Character 0 in any encoding must be called '.notdef', and represent a null character.
- Other null characters in the base encoding must be called '.NotDef', to distinguish them from character 0.
- Non-base encoding files wanting to refer to the null character should use '.notdef' in all cases.
- The other character names should follow the Adobe PostScript names wherever possible. (See *PostScript Language Reference Manual*. Adobe Systems Incorporated (1990) 2nd ed. Addison-Wesley, Reading, Mass, USA.) This is to enable the encoding to refer to Adobe character names when included as part of a print job by the PostScript printer driver.
- The number of characters described by the base encoding can be anything from 0 to 768, and should refer to distinct characters (apart from the '.NotDef's). Other encodings, however, must contain exactly 256 characters, which need not be distinct.

Font Messages files

The format of font Messages files is the same as that of ordinary message files, as documented in the chapter entitled *MessageTrans* on page 3-745, with those exceptions detailed below.

The valid tokens are:

Encoding_	encoding (based on a base encoding)
BEncoding_	base encoding (eg Base0)
Font_	font which doesn't vary with alphabet (eg Symbol font)
LFont_	font which does vary with alphabet (a 'language' font)

The tokens are of the form 'Font_' followed by the identifier of the font in the font directory, and their values are the names of those fonts. If the value is null, then the font name is taken to be the same as the identifier.

The values of the encoding tokens should normally be null, but you **must** define them for all encodings within the directory holding the Messages file if you want to use a font that references them. Also, you must not prefix the base encoding id with '/' even though its filename is '/Basen'. This is because the '/' in the filename is only used by Font_ListFonts when it is scanning a font directory to determine base encodings from encodings.

Identifiers should use characters in the range &20 - &7E, to aid in international portability. However, the font names should use the alphabet of the relevant territory, as determined by the country number on the end of the message file name.

Within a font name, the following characters are special:

- . The first dot encountered causes the font to be split over two menu levels. Subsequent dots do not cause further submenu splitting.
- * An asterisk as the last character of a font name is not treated as part of the name, but marks this font as being the default for that family. Clicking on the menu entry for the font family will select the default weight and/or style for the family, even though the font weights and styles are in a subdirectory. This is normally *fontfamily*.Medium, but there are other examples (eg Selwyn).

Note that if a font name is given as '*' alone, then the name is the same as the identifier and it is also made the default for that family.

For example, a ‘Messages1’ file for the ROM fonts might be:

```
BEncoding_Base0:  
Encoding_Latin1:  
Encoding_Latin2:  
Encoding_Latin3:  
Encoding_Latin4:  
LFont_Corpus.Bold:  
LFont_Corpus.Bold.Oblique:  
LFont_Corpus.Medium:*  
LFont_Corpus.Medium.Oblique:  
LFont_Homerton.Bold:Helvetica bold  
LFont_Homerton.Bold.Oblique:Helvetica bold oblique  
LFont_Homerton.Medium:Helvetica*  
LFont_Homerton.Medium.Oblique:Helvetica oblique  
LFont_Trinity.Bold:  
LFont_Trinity.Bold.Italic:  
LFont_Trinity.Medium:*  
LFont_Trinity.Medium.Italic:
```

This aliases the Homerton font family so that users see it named ‘Helvetica’, and sets the default font in each family to the one of ‘Medium’ weight.

Music files

Header

Size	Description
8	'Maestro' followed by linefeed (&0A)
1	2 (type 2 music file)

This is followed by 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
5	n = number of bytes in the 'Gates' array (stored as a BASIC integer – ie &40 followed by four bytes of data, most significant first).
5×8	$q1 \dots q8$ = number of bytes in queue of notes and rests for each of the 8 channels 1...8 (stored as BASIC integers – ie &40 followed by four bytes of data, most significant first).
n	gate data (see <i>Gates</i> on page 4-494) <i>For c = 1 to 8 (ie for each channel in turn)</i>
$\Sigma q1 \dots q8$	data for all notes or rests in channel c (see <i>Notes and rests</i> on page 4-496) <i>Next c</i>

Stave data

Size	Description
1	2 indicates Stave data follows
1	number of music staves – 1 (0 - 3)
1	number of percussion staves (0 - 1)

Which channels are used by which staves depends on the number of music staves and the number of percussion staves as follows:

Music staves	Percussion staves	Stave 1	Stave 2	Stave 3	Stave 4	Percussion
1	0	1 - 8				
1	1	1 - 7				8
2	0	1 - 4	5 - 8			
2	1	1 - 4	5 - 7			8
3	0	1	2 - 5	6 - 8		
3	1	1	2 - 5	6, 7		8
4	0	1, 2	3, 4	5, 6	7, 8	
4	1	1, 2	3, 4	5, 6	7	8

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:

Size	Description
1	channel number (always consecutive 1 - 8)
1	<i>voice number</i> : 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 (0 - 6 = 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, or 210beats per minute

To convert to values to program into SWI Sound_QTempo, use the formula:

$$\text{Sound_QTempo value} = \text{beats per minute} \times 128 \times 4096 / 6000$$

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
$\Sigma n1 \dots n8$	8 null terminated strings for each <i>voice number</i> used in ascending order in command 3 above.

MIDI channels

Size	Description
1	9 indicates MIDI channel numbers follow
1×8	MIDI channel number on this staff (0 \Rightarrow not transmitted over MIDI, else 1 - 16); one byte for each channel

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, or 2 bytes for an attribute such as a time signature, key signature, bar line, clef, etc.

Note or rest

A note or rest is represented by a single non-zero byte.

Bit(s)	Description
0 - 7	Gate mask: bit n set \Rightarrow gate 1 note or rest from queue n .

Attribute

An attribute is represented by a null byte (so that it can be distinguished from a note or rest), followed by a byte describing the attribute.

Byte	Description
0	0
1	one of the following forms:

Time signature

Bit(s)	Description
0	1
1 - 4	number of beats per bar – 1 (0 - 15)
5 - 7	beat type (0 = breve, to 7 = hemidemisemi-quaver)

Key signature

Bit(s)	Description
0 - 1	10 binary (ie bit 1 set)
2	type of accidental (0 = sharp, 1 = flat)
3 - 5	number of accidentals in key signature (0 - 7)
6 - 7	reserved (must be zero)

Clef

Bit(s)	Description
0 - 2	100 binary (ie bit 2 set)
3 - 4	0 = treble, 1 = alto, 2 = tenor, 3 = bass
5	reserved (must be zero)
6 - 7	stave – 1 (0 - 3)

Slur

Bit(s)	Description
0 - 3	1000 binary (ie bit 3 set)
4	1 = on, 0 = off
5	reserved (must be zero)
6 - 7	stave – 1 (0 - 3)

Octave shift

Bit(s)	Description
0 - 4	10000 binary (ie bit 4 set)
5	0 = up, 1 = down
6 - 7	stave – 1 (0 - 3)

Bar

Bit(s)	Description
0 - 5	100000 binary (ie bit 5 set)
6 - 7	reserved (must be zero)

Reserved for future expansion

Bit(s)	Description
0 - 6	1000000 binary (ie bit 6 set)
0 - 7	10000000 binary (ie bit 7 set)

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 = up, 1 = down)
1	1 ⇒ join beams (barbs) to next note
2	1 ⇒ tie with next note
3 - 7	stave line position (1 - 31, 16 = centre line)
8 - 10	accidental: <ul style="list-style-type: none"> 0 = no accidental 1 = natural 2 = sharp 3 = flat 4 = double-sharp 5 = double-flat 6 = natural sharp 7 = natural flat
11 - 12	number of dots (0 - 3)
13 - 15	type (0 = breve, to 7 = hemidemisemiquaver)

Rests

Bits	Description
0 - 10	reserved (set to zero)
11 - 12	number of dots (0 - 3)
13 - 15	type (0 = breve, to 7 = hemidemisemiquaver)

If a rest coincides with a note, its position is determined by the following note on the same channel.

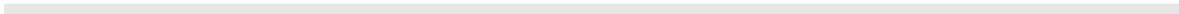
Squash files

Squash files are generated by the !Squash application, which in turn uses the Squash module, as documented in the chapter entitled *Squash* on page 4-103.

Squash files consist of a small fixed size header put in by !Squash, followed by compressed data produced by the Squash module. The header has the following format:

```
typedef struct
{
    char id[4];                /* Should be "SQSH" */
    unsigned int length;
    unsigned int load;
    unsigned int exec;
    int reserved;             /* Should be zero */
} squash_header;
```

The length, load and exec are the file length, load address and execution address of the original file before it was compressed (although the load and exec typically hold the filetype and date/time stamp). If the id is not SQSH, then the rest of the file is not in the same format.



This appendix details standard variables used in RISC OS, and gives important guidelines on the names you should use for any system variables you create for your applications to use.

Application variables

The following section gives standard names used for variables that are bound to a particular application. An application should not need to set all these variables, but where one of the variables below matches your needs, you should use it and follow the given guidelines. Where you need a system variable and can't find a relevant one below, you should use your own, naming it *App\$...*

In the descriptions below you should replace *App* with your application's name. You must first register this name with Acorn, to avoid any possibility of your system variables clashing with those used by other programmers' applications; see *Appendix H: Registering names* on page 4-549.

App\$Dir

An *App\$Dir* variable gives the full pathname of the directory that holds the application *App*. This is typically set in the application's !Run file by the line:

```
Set App$Dir <Obey$Dir>
```

App\$Path* and *App\$Path_Message

An *App\$Path* variable gives the full pathname of the directory that holds the application *App*. An *App\$Path* variable differs from an *App\$Dir* variable in two important respects:

- The pathname includes a trailing '.'
- The variable may hold a set of pathnames, separated by commas.

An *App\$Path_Message* variable gives an alternative error message to be used if the path *App*: cannot be found. This message is then used instead of the default one provided by RISC OS.

It's common to use an *App\$Dir* variable rather than an *App\$Path* variable, but there may be times when you need the latter.

An *App\$Path* variable might, for example, be set in the application's !Run file by the line:

```
Set App$Path <Obey$Dir>.,%.App.
```

if the application held further resources in the subdirectory *App* of the library.

App\$Options

An *App\$Options* variable holds the start-up options of the application *App*:

- An option that can be either on or off should consist of a single character, followed by the character '+' or '-' (eg M+ or S-).
- Other options should consist of a single character, followed by a number (eg P4 or F54).
- Options should be separated by spaces; so a complete string might be F54 M+ P4 S-.

This variable is typically used to save the state of an application to a desktop boot file, upon receipt of a desktop save message. A typical line output to the boot file might be:

```
Set App$Options F54 M+ P4 S-
```

You should only save those options that differ from the default, and hence not output a line at all if the application is in its default state. You should however be prepared to read options that set the default values, in case users explicitly add such options.

App\$PrintFile

An *App\$PrintFile* variable holds the name of the file or system device to which the application *App* prints. Typically this will be printer:, and would be set in your application's !Run file as follows:

```
Set App$PrintFile printer:
```

App\$Resources

An *App\$Resources* variable gives the full pathname of the directory that holds the application *App*'s resources. This might be set in the application's !Run file by the line:

```
Set App$Resources App:Resources
```

Note the use of *App*: to make use of *App\$Path*.

App\$Running

An *App\$Running* variable shows that the application *App* is running. It should have the value 'Yes' if the application is running. This might be used in the application's !Run file as follows:

```
If "App$Running" <> "" then Error App is already running
Set App$Running Yes
```

When the application stops running, you should use *Unset to delete the variable.

Changing and adding commands

Alias\$Command

An *Alias\$Command* variable is used to define a new command named *Command*. For example:

```
Set Alias$Mode echo |<22>|<%0>
```

By using the name of an existing command, you can change how it works.

FileSwitch variables

FileSwitch\$...

FileSwitch\$CurrentFilingSystem contains the name of the current filing system, and *FileSwitch\$TemporaryFilingSystem* contains the name of the temporary filing system. *FileSwitch\$SpecialField* contains the last special field to have been evaluated as a path was processed. See also the section entitled *Using FileSwitch\$SpecialField with path variables* on page 2-20.

FileSwitch\$FilingSystem\$...

Most filing systems provide system variables used to store their currently selected directory, previously selected directory, library directory, and user root directory. For a filing system *fs*, these are respectively *FileSwitch\$fs\$CSD*, *FileSwitch\$fs\$PSD*, *FileSwitch\$fs\$Lib* and *FileSwitch\$fs\$URD*.

Using file types

File\$Type_XXX

A File\$Type_XXX variable holds the textual name for a file having the hexadecimal file type XXX. It is typically set in the !Boot file of an application that provides and edits that file type. For example:

```
Set File$Type_XXX TypeName
```

The reason the !Boot file is used rather than the !Run file is so that the file type can be converted to text from the moment its 'parent' application is first seen, rather than only from when it is run.

Alias\$@LoadType_XXX, Alias\$@PrintType_XXX and Alias\$@RunType_XXX

These variables set the commands used to respectively load, print and run a file of hexadecimal type XXX. They are typically set in the !Boot file of an application that provides and edits that file type. For example:

```
Set Alias$@PrintType_XXX /<Obey$Dir> -Print  
Set Alias$@RunType_XXX /<Obey$Dir>
```

Note that the above lines **both have a trailing space** (invisible in print!).

The reason the !Boot file is used rather than the !Run file is so that files of the given type can be loaded, printed and run from the moment their 'parent' application is first seen, rather than only from when it is run.

For more information see the section entitled *Load-time and run-time system variables* on page 2-17.

Absent filing systems

FilingSystem\$Path_Message

A FilingSystem\$Path_Message variable gives an alternative error message to be used if the *FilingSystem* cannot be found. This message is then used instead of the default one provided by RISC OS.

Setting the command line prompt

CLI\$Prompt

The CLI\$Prompt variable sets the command line interpreter prompt. By default this is ‘*’. One common way to change this is so that the system time is displayed as a prompt. For example:

```
SetMacro CLI$Prompt <Sys$Time> *
```

This is set as a macro so that the system time is evaluated each time the prompt is displayed.

Configuring RISC OS commands

Copy\$Options, Count\$Options and Wipe\$Options

These variables set the behaviour of the *Copy, *Count and *Wipe commands. For a full description, see page 2-154, page 2-157 and page 2-195 respectively.

System path variables

File\$Path and Run\$Path

These variables control where files are searched for during, respectively, read operations or execute operations. They are both path variables, which means that – in common with other path variables – they consist of a comma separated list of full pathnames, each of which has a trailing ‘.’.

If you wish to add a pathname to one of these variables, you must ensure that you append it once, and once only. For example, to add the ‘bin’ subdirectory of an application to Run\$Path, you could use the following lines in the application’s !Boot file:

```
If "<App$Path>" = "" then Set Run$Path <Run$Path>,<Obey$Dir>.bin.  
Set App$Path <Obey$Dir>.
```

For more information see the section entitled *File\$Path and Run\$Path* on page 2-18.

Obey files

Obey\$Dir

The Obey\$Dir variable is set to the directory from which an Obey file is being run, and may be used by commands within that Obey file. For examples, see various other sections of this chapter. For more detailed information, see the section entitled *Obey\$Dir* on page 4-352.

Time and date

Sys\$Time, Sys\$Date and Sys\$Year

These variables are code variables that are evaluated at the time of their use to give, respectively, the current system time, date and year.

For an example of the use of Sys\$Time, see the section entitled *CLI\$Prompt* on page 4-503.

Sys\$DateFormat

The Sys\$DateFormat variable sets the format in which the date is presented by the SWI OS_ConvertStandardDateAndTime (see page 1-449). For details of the format used by this variable, see the section entitled *Format field names* on page 1-414.

Return codes

Sys\$ReturnCode, Sys\$RCLimit

The Sys\$ReturnCode variable contains the last return value given by the SWI OS_Exit, and the Sys\$RCLimit variable sets the maximum return value that will not generate an error. For more details, see page 1-305.

!System and !Scrap

System\$Dir and System\$Path

These variables give the full pathname of the System application. They have the same value, save that System\$Path has a trailing '.', whereas System\$Dir does not. You must not change their values.

(There are two versions of this pathname for reasons of backward compatibility.)

Wimp\$Scrap

The Wimp\$Scrap variable gives the full pathname of the Wimp scrap file used by the file transfer protocol. You must not use this variable for any other purpose, nor change its value.

Wimp\$ScrapDir

The Wimp\$ScrapDir variable gives the full pathname of a scrap directory within the Scrap application, which you may use to store temporary files. You must not use this variable for any other purpose, nor change its value.

The desktop

Desktop\$File

The Desktop\$File variable shows the desktop boot file that was used to start the desktop.

Wimp\$State

The Wimp\$State variable shows the current state of the Wimp. If the desktop is running, it has the value 'desktop'; otherwise it has the value 'commands'.

The Task Window

TaskWindow\$Server

The TaskWindow\$Server variable gives the pathname of the application used to start up task windows.

Setting default options for devices

DeviceFS\$Device\$Options

The DeviceFS\$Device\$Options variable holds default options for a DeviceFS device. For more details see the chapter entitled *DeviceFS* on page 2-429.

Setting paths for printing

PrinterType\$n

A PrinterType\$n variable contains the path used to print to printer type *n*. For example:

***Show PrinterType\$0**

PrinterType\$0 : null:

Introduction

This appendix describes version 1.00 of the *Acorn Terminal Interface Protocol* (or *Acorn TIP*) used to communicate between a terminal emulator and a protocol module. By using this protocol you can integrate your own terminal emulators and protocol modules with those provided by the TCP/IP Protocol Suite.

Although this chapter only talks about the Acorn TIP in the context of terminal emulators and protocol modules, there's no reason why you shouldn't use it for other applications that involve input and output.

Protocol modules

A *protocol module* converts one of the many different protocols computers use for input and output to the Acorn TIP. For example in the case of the VT220 application and the protocol modules supplied as part of the TCP/IP Protocol Suite, we have:

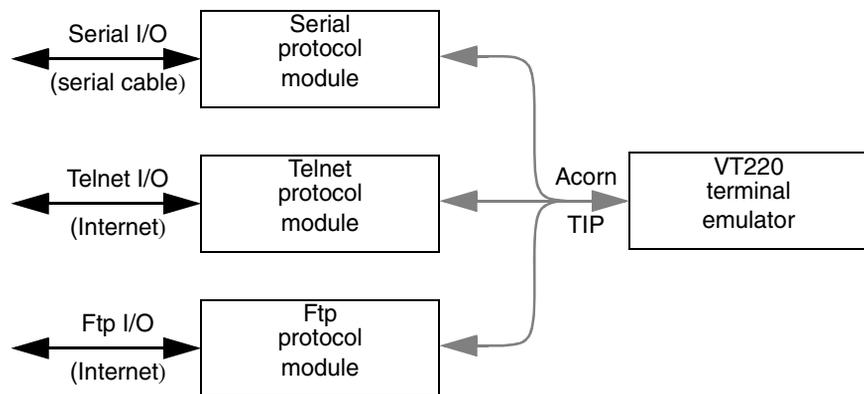


Figure 92.1 Structure of the VT220 module and protocol modules

- Data passing between a terminal emulator and a protocol module uses the Acorn TIP, and passes over a *logical link*. These are grey in the drawing above.

- Data passing between a protocol module and a remote machine or process uses whatever protocol the module is designed to support, and passes over a *connection*. These are black in the drawing above.

Using the Acorn TIP

If you decide to write other protocol modules and/or terminal emulators, you should use the Acorn TIP. Since this provides a standard interface between protocol modules and terminal emulators, users will be able to use your modules and emulators with the TCP/IP ones, and with ones that other programmers write too. If your software is compatible, we think it's more likely users will buy it.

Writing a protocol module

If you're writing a protocol module, you must first familiarise yourself with how a RISC OS relocatable module works. You'll find full details of this in the chapter entitled *Modules* on page 1-203. Your protocol module must conform to the standards laid out in that chapter.

Service calls

You must support the service calls detailed in this chapter.

SWIs

You must also support various SWIs from the set detailed in this chapter. These must be at the defined offsets from your module's SWI base number, which is allocated by Acorn. To support many of these SWIs you will need to send suitable commands over the physical connection to the remote host.

- You must support:

Offset	SWI name
0	Protocol_OpenLogicalLink
1	Protocol_CloseLogicalLink
2	Protocol_GetProtocolMenu
3	Protocol_OpenConnection
4	Protocol_CloseConnection
7	Protocol_MenuItemSelected
8	Protocol_UnknownEvent
9	Protocol_GetLinkState
10	Protocol_Break

- If your protocol module supports the sending of data over a connection to a remote machine (or process) you must also support:

Offset	SWI Name
5	Protocol_TransmitData

If you have chosen to support file transfer SWIs you must furthermore support:

Offset	SWI Name
11	Protocol_SendFile
12	Protocol_SendFileData
13	Protocol_AbortTransfer

- If your protocol module supports the receipt of data over a connection from a remote machine (or process) you must also support:

Offset	SWI Name
6	Protocol_DataRequest

If you have chosen to support file transfer SWIs you must furthermore support:

Offset	SWI Name
13	Protocol_AbortTransfer
14	Protocol_GetFileInfo
15	Protocol_GetFileData
17	Protocol_GetFile

- You may also choose to support:

Offset	SWI Name
18	Protocol_DirOp

Data structures

Your protocol module must keep two different types of data structure constantly updated, as terminal emulators may directly access these any time they need to. These are:

- A single *protocol information block* which contains the following information:

Offset	Information
0	pointer to protocol name string
4	pointer to protocol version string
8	pointer to protocol copyright string
12	maximum number of connections allowed by module
16	current number of open connections

The three strings are all null-terminated, and have a maximum length of 30 characters. For more details see *Protocol_OpenLogicalLink (Offset 0)* on page 4-518.

- A *poll word* for each logical link that shows the status of that link by the state of various bit flags:

Bit	Meaning when set
0	data is pending
1	file is pending
2	paused operation is to continue

For more details see *Protocol_OpenConnection (Offset 3)* on page 4-522.

Multiple links and connections

All protocol modules **must** (if physically possible) support multiple logical links, and multiple connections.

Writing a terminal emulator

If you're writing a terminal emulator there are various functions that it's likely you'll want it to support. This section tells you which SWIs you'll need to use for many such functions, and outlines how to use them. The later section that details each SWI will give you the detailed information you need.

Finding available and compatible protocols

To find what protocols are available and compatible with the needs of your emulator, you must repeatedly issue `Service_FindProtocols` (page 4-514) until it is not claimed. Then you must issue `Service_FindProtocolsEnd` (page 4-516).

Choosing a protocol and opening a link

For your user to choose a protocol, you'll probably want to give them a menu of the ones you found to be available. Once they've made the choice, you can then issue `Service_ProtocolNameToNumber` (page 4-517) to find the base SWI number of their chosen protocol module. You can then use this base number to call the SWI `Protocol_OpenLogicalLink` (page 4-518), since its offset from the base number you just found is zero.

You can also use the facilities outlined in the section entitled *Protocol modules and the Wimp* on page 4-512 to provide menus so that your user can set up the way the protocol and connection will work.

Opening a connection

To open a connection, call `Protocol_OpenConnection` (page 4-522). Sometimes the protocol module won't immediately be able to open the connection; you'll need to use `Protocol_GetLinkState` (page 4-533) to find out whether the connection eventually makes or fails.

Closing a connection and a link

To close a connection, call `Protocol_CloseConnection` (page 4-525). To close a logical link, call `Protocol_CloseLogicalLink` (page 4-520); this also closes any associated connections.

Examining the poll word

When you open a connection, you set the address of a poll word. The protocol module sets bits in this word when it needs attention. It's vital that your emulator regularly examines this word so that the protocol module gets adequate service. We suggest you do so each time you get a null event from `Wimp_Poll`.

Sending data

To send data, call `Protocol_TransmitData` (page 4-526).

Receiving data

When the protocol module receives data over a connection, it will notify your emulator by setting a bit in the poll word. To get the data forwarded to your emulator, call `Protocol_DataRequest` (page 4-528).

Sending files

To send a file, call `Protocol_SendFile` (page 4-537) to give details of the file to the protocol module. When the protocol module shows it is ready for you to send the file (by using the poll word), send the file in one or more data packets by repeatedly calling `Protocol_SendFileData` (page 4-539). Finally, call `Protocol_SendFileData` (page 4-539) a last time to mark the end of the file transfer.

You can use this call to send multiple files.

Wherever possible you should make sure that the data packets are small enough that they can be quickly sent, so your emulator doesn't hog the computer for long periods.

Receiving files

When the protocol module receives a file over a connection, it will notify your emulator by setting a bit in the poll word. To get the file forwarded to your emulator, call `Protocol_GetFileInfo` (page 4-542) to get details of the file. When the protocol module shows it is ready to forward the file (again by using the poll word), call `Protocol_GetFileData` (page 4-543) until you've received all the data packets making up the file.

Explicitly getting a file

To explicitly get a file, call `Protocol_GetFile` (page 4-546). You'll actually receive it just as we outlined above.

Aborting file operations

To abort any file operation, call `Protocol_AbortTransfer` (page 4-541).

Directory operations

There are no SWIs specified in the Acorn TIP to send, receive or get entire directories in one call. Instead we provide a single SWI call – `Protocol_DirOp` (page 4-547) – with which you can create a directory, move into a directory, and move one level up a directory tree. You can combine this SWI with the ones outlined above to move around a remote file system, creating directories, and sending and getting files at will (subject, of course, to your having access rights).

Protocol modules and the Wimp

The Acorn TIP provides several calls which help interaction between the Wimp and protocol menus. These are necessary because the 'pick and mix' nature of protocol modules and terminal emulators means you'll have to combine menus from each; and because protocol modules are not foreground tasks, and so don't receive notice of menu selections and Wimp events.

To get a protocol's menu tree, call `Protocol_GetProtocolMenu` (page 4-521); you can then combine it with your emulator's menu tree. If a user clicks on the protocol module's part of the menu tree, call `Protocol_MenuItemSelected` (page 4-530) to pass this on. To pass on a Wimp event to a protocol module, call `Protocol_UnknownEvent` (page 4-532); you should do this for every event your emulator can't deal with, as the protocol module may be able to.

Generating a break

Finally, you can generate a Break over the connection by calling `Protocol_Break` (page 4-535).

Documentation of Service Calls and SWIs

The rest of this chapter details in turn each Service Call and SWI used to communicate between a protocol module and a terminal emulator. It looks at each in three stages:

- 1 What your terminal emulator should do before calling the Service Call or SWI.
- 2 What a protocol module should do when it receives the Service Call or SWI.
- 3 What your terminal emulator should do when the call returns to it.

We've followed the same viewpoint throughout as we have above: we assume that you're writing a terminal emulator to work with someone else's protocol module. So we talk about **your** terminal emulator, but **the** protocol module. If, in fact, you're writing a protocol module, you should find it easy enough to make the necessary shift of viewpoint.

Service Calls

Service_FindProtocols (Service Call &41580)

Finds all available compatible protocols

On entry

R1 = &41580 (reason code)
 R2 = lowest TIP version supported $\times 100$ (first public version was 1.00)
 R3 = last TIP version known $\times 100$ (current version is 1.00)
 R4 = emulator flags

On exit

R1 = 0 to claim, else registers preserved to pass on
 R2 = pointer to protocol name string (null terminated)
 R3 = base SWI number of protocol module
 R4 = pointer to protocol information block
 R5 = protocol flags

Use

Use this service call in your **terminal emulator** to find all available compatible protocol modules. (For full details of OS_ServiceCall see page 1-256.) You should:

- 1 Repeatedly issue this service call until it is not claimed – without polling the Wimp in the meantime.
- 2 Issue Service_FindProtocolsEnd (see page 4-516).

The emulator flags have the following meanings:

Bits	Value	Meaning
0	0	emulator doesn't support file transfer calls
	1	emulator supports file transfer calls
1-2	00	direction of link immaterial
	01	one-way link wanted – protocol to emulator
	10	one-way link wanted – emulator to protocol
	11	two-way link needed
3	0	bits 1-2 are minimum requirement
	1	bits 1-2 are exact requirement

All other bits are reserved and must be zero.

The **protocol module** checks to see if:

- it uses a version of the Acorn TIP in the range supported by the terminal emulator
- it supports links in the direction required by the terminal emulator.

If one of the above isn't true, the protocol module must not claim the call – that is, it must return with registers preserved.

If both the above are true it must claim the call – that is, it must return with the values shown above in the section entitled *On exit*. It must then set an internal flag so it doesn't claim this call again until it receives a `Service_FindProtocolsEnd`.

The protocol information block it returns contains the following information:

Offset	Information
0	pointer to protocol name string
4	pointer to protocol version string
8	pointer to protocol copyright string
12	maximum number of connections allowed by module
16	current number of open connections

The three strings are all null-terminated, and have a maximum length of 30 characters. The protocol module must always keep this block updated so terminal emulators can directly access it.

The protocol flags it returns have the following meanings:

Bits	Value	Meaning
0	0	can open new link
	1	can't open new link, or not useful (see below)
1	0	protocol doesn't support file transfer SWIs
	1	protocol supports file transfer SWIs
2	0	protocol doesn't support <code>Protocol_DirOp</code>
	1	protocol supports <code>Protocol_DirOp</code>

If the protocol is mainly for file transfer (such as Ftp) and the terminal emulator doesn't support file transfer calls (bit 0 of R3 was clear on entry) the protocol module should set bit 0 to show it's 'not useful'.

All other bits are reserved and must be zero.

Related Service Calls

`Service_FindProtocolsEnd` (page 4-516),
`Service_ProtocolNameToNumber` (page 4-517)

Service_FindProtocolsEnd (Service Call &41581)

Indicates that protocol modules must again respond to Service_FindProtocols

On entry

R1 = &41581 (reason code)

On exit

R1 = 0 to claim, else preserved to pass on

Use

Use this service call in your **terminal emulator** to indicate the end of your search for available protocols.

Protocol modules must change their internal flag so they respond again to Service_FindProtocols calls – from whatever terminal emulator the calls originate. They **must not** claim this call.

Related Service Calls

Service_FindProtocols (page 4-514),

Service_ProtocolNameToNumber (page 4-517)

Service_ProtocolNameToNumber (Service Call &41582)

Requests the conversion of a protocol name to a base SWI number

On entry

R1 = &41582

R2 = pointer to protocol name (null-terminated)

On exit

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

R2 = base SWI number for protocol

Use

Use this service call in your **terminal emulator** to request the conversion of a protocol name to a base SWI number.

If a **protocol module** recognises the protocol name it must claim the call and return the base SWI number of the protocol. Otherwise it must pass the call on.

Related Service Calls

Service_FindProtocols (page 4-514),

Service_FindProtocolsEnd (page 4-516)

SWI calls

Protocol_OpenLogicalLink (Offset 0)

Opens a logical link to a protocol module

On entry

R0 = terminal emulator's link handle
R1 = pointer to terminal identifier string (null terminated)

On exit

R0 = protocol module's link handle
R1 = protocol module's Wimp_Poll mask
R2 = pointer to protocol information block
R3 = protocol information flags

Use

Use this call in your **terminal emulator** to open a logical link to a protocol module. The handle you pass on entry will be returned to you by future SWI calls you make to the protocol module – we suggest you use a pointer to your data structures that are specific to this link.

You may use the terminal identifier string for such things as setting the 'type' of your terminal emulator on the remote machine.

The **protocol module** returns its own handle for the link – again this is typically a pointer to its own data that is specific to the link. The Wimp_Poll mask it returns specifies those Wimp events that it doesn't need.

The protocol information block contains the following information:

Offset	Information
0	pointer to protocol name string
4	pointer to protocol version string
8	pointer to protocol copyright string
12	maximum number of connections allowed by module
16	current number of open connections

The three strings are all null-terminated, and have a maximum length of 30 characters. The protocol module must always keep this block updated so terminal emulators can directly access it.

The protocol information flags have the following meanings:

Bit	Meaning when set
0	protocol needs more information to open a connection
1	protocol supports file transfer SWIs
2	protocol supports Protocol_DirOp

All other bits are reserved and must be zero.

When this call returns to your **terminal emulator** you should examine bit 0 of the protocol information flags. If it is clear then you should immediately call Protocol_OpenConnection; if it is set you will have to wait until the user shows they are ready to supply the information the protocol module needs (by, for instance, moving the pointer over the arrow that shows an ‘open connection’ menu item to have a submenu).

Also, you should AND the protocol module’s Wimp_Poll mask with your terminal emulator’s own one. Use the resultant mask whenever you call Wimp_Poll.

Related SWIs

Protocol_CloseLogicalLink (page 4-520), Protocol_OpenConnection (page 4-522), Protocol_CloseConnection (page 4-525), Protocol_GetLinkState (page 4-533)

Protocol_CloseLogicalLink (Offset 1)

Closes a logical link to a protocol module

On entry

R0 = protocol module's link handle

On exit

R0 preserved

Use

Use this call in your **terminal emulator** to close a logical link to a protocol module.

The **protocol module** closes any connections that are associated with the logical link.

Related SWIs

Protocol_OpenLogicalLink (page 4-518), Protocol_OpenConnection (page 4-522),
Protocol_CloseConnection (page 4-525), Protocol_GetLinkState (page 4-533)

Protocol_GetProtocolMenu (Offset 2)

Gets a protocol's menu tree

On entry

R0 = protocol module's link handle

On exit

R0 = terminal emulator's link handle

R1 = pointer to protocol and link specific Wimp menu block
(as used by Wimp_CreateMenu)

Use

Use this call in your **terminal emulator** to get a protocol's menu tree. You must use this call each time you want to open the protocol's menu, as it may change depending on the state of the logical link. For example items may become unavailable and so be greyed out, or the user may change the contents of a writable entry.

The **protocol module** returns a pointer to a menu block that is the same as that used by Wimp_CreateMenu. (See page 3-153 for details of this call.) This menu block must accurately reflect the current state of the logical link between the terminal emulator and the protocol module.

Related SWIs

Protocol_MenuItemSelected (page 4-530), Protocol_UnknownEvent (page 4-532)

Protocol_OpenConnection (Offset 3)

Opens a connection from a protocol module

On entry

R0 = protocol module's link handle
R1 = pointer to poll word for this connection
R3 = pointer to protocol specific string (null-terminated), or 0
R4 = x coordinate of top-left corner of dialogue box
R5 = y coordinate of top-left corner of dialogue box

On exit

R0 = terminal emulator's link handle
R1 = pointer to connection name (null-terminated)
R2 = pointer to protocol specific information, or 0
R3 = protocol status flags

Use

Use this call in your **terminal emulator** to open a connection from a protocol module. At the same time you pass the protocol module the address of a poll word in your workspace, which your terminal emulator must regularly check to review the state of the logical link to the protocol module. We suggest you do so each time you get a null event from Wimp_Poll.

When a bit is set in the poll word, something needs attention. The table below shows the meaning of each bit, and the **initial** SWI call you have to make to handle the situation. See the relevant pages for details of what to do, and of any further calls you may need to make.

Bit	Meaning when set	Call needed
0	data is pending	Protocol_DataRequest
1	file is pending	Protocol_GetFileInfo
2	paused operation is to continue	Protocol_GetFileData or Protocol_SendFileData or Protocol_DirOp

The poll word must be in RMA space, so the protocol module can update it whether or not your terminal emulator is the foreground task.

The values you need to pass in R3, R4 and R5 depend on circumstances:

- If the protocol module needs no further information to open the connection these values are ignored.
- If the user has shown they are ready to supply the information the protocol module needs (typically by moving the pointer over the arrow that shows an ‘open connection’ menu item to have a submenu), you must set R3 to zero, and R4 and R5 to the coordinates where you want the protocol module to open a dialogue box. You can get these coordinates by making your terminal emulator’s menu issue `Message_MenuWarning` when the submenu is to be activated (see `Wimp_CreateMenu` on page 3-153 and `Wimp_SendMessage` on page 3-193).
- If the user has already supplied you with the information that the protocol module needs (say in a script) you should pass that in R3. The values of R4 and R5 are ignored.

The **protocol module** opens the connection after first (if necessary) using a dialogue box to get any information it needs.

The documentation of a protocol module **must** state the format of information it expects to find in R3 (if it needs any). Wherever possible, this format should consist of the same fields that the protocol module provides in its dialogue box, in the same order, and comma-separated.

The protocol module returns a connection name suitable for the terminal emulator to use as a window title (if the connection is open or pending). The protocol specific information it returns may be used for error messages. The protocol status flags it returns have the following meanings:

Bits	Value	Meaning
0-1	00	no connection opened
	01	connection pending
	10	connection open
	11	connection failed
2	0	no data pending
	1	data pending

All other bits are reserved and must be zero. The protocol module should select ‘connection failed’ in preference to ‘no connection opened’.

When this call returns to your **terminal emulator** you must examine the state of these flags:

- If the connection failed (bits 0 and 1 are set) and no data is pending (bit 2 is clear) you must attempt to close the connection by calling `Protocol_CloseConnection`.

- If the connection is pending you must wait until bit 0 of the logical link's poll word is set. Then you should call `Protocol_GetLinkState` to find if the connection was opened, or if it failed.
- Bit 2 ('data pending') has exactly the same meaning as bit 0 of a logical link's poll word, and is provided to reduce the amount of polling that needs to be done. If it is set you should initiate the data transfer by calling `Protocol_DataRequest`.

Related SWIs

`Protocol_OpenLogicalLink` (page 4-518), `Protocol_CloseLogicalLink` (page 4-520),
`Protocol_CloseConnection` (page 4-525), `Protocol_GetLinkState` (page 4-533)

Protocol_CloseConnection (Offset 4)

Closes a link's connection from a protocol module

On entry

R0 = protocol module's link handle

On exit

R0 = pointer to protocol specific information, or 0

Use

Use this call in your **terminal emulator** to close a link's connection from a protocol module.

The **protocol module** closes the connection associated with the given link.

Related SWIs

Protocol_OpenLogicalLink (page 4-518), Protocol_CloseLogicalLink (page 4-520), Protocol_OpenConnection (page 4-522), Protocol_GetLinkState (page 4-533)

Protocol_TransmitData (Offset 5)

Transmits data over a connection via a protocol module

On entry

R0 = protocol module's link handle
R1 = pointer to receive buffer
R2 = length of receive buffer (in bytes)
R3 = pointer to transmit buffer
R4 = length of transmit buffer (in bytes)
R5 = emulator transmit flags

On exit

R0 = terminal emulator's link handle
R2 = number of bytes of data placed in receive buffer
R3 = protocol status flags
R4 = pointer to protocol specific information

Use

Use this call in your **terminal emulator** to transmit data over a connection via a protocol module. You'll also receive any pending data that the protocol module has been holding for you.

The emulator transmit flags have the following meanings:

Bit	Value	Meaning
3	0	transmitted data is in bytes
	1	transmitted data is in words

All other bits are reserved and must be zero. If the transmitted data is in words, each word contains one character in the least significant byte.

The **protocol module** transmits the data over the connection. Also, if it has any pending data for the terminal emulator it forwards as much as it is able to place in the emulator's receive buffer.

The protocol specific information it returns may be used for error messages.

The protocol status flags it returns have the following meanings:

Bits	Value	Meaning
0-1	00	no connection opened
	01	connection pending
	10	connection open
	11	connection failed
2	0	no data pending
	1	more data pending
3	0	data is in bytes
	1	data is in words

All other bits are reserved and must be zero.

When this call returns to your **terminal emulator** you must check R2 to see if you have received any data, and process it if necessary. You must also examine the protocol status flags in R3:

- If the connection failed (bits 0 and 1 are set) and no data is pending (bit 2 is clear) you must attempt to close the connection by calling `Protocol_CloseConnection`.
- If the connection is pending you have made an error in your programming by trying to use the connection before it has been properly opened.
- Bit 2 ('more data pending') has exactly the same meaning as bit 0 of a logical link's poll word, and is provided to reduce the amount of polling that needs to be done. If it is set you should initiate the data transfer by calling `Protocol_DataRequest`.
- If the data you've received is in words, each word contains one character in the least significant byte.

Related SWIs

`Protocol_SendFile` (page 4-537), `Protocol_SendFileData` (page 4-539)

Protocol_DataRequest (Offset 6)

Requests that a protocol module forwards any pending data

On entry

R0 = protocol module's link handle
R1 = pointer to receive buffer
R2 = length of receive buffer (in bytes)

On exit

R0 = terminal emulator's link handle
R1 preserved
R2 = number of bytes of data placed in receive buffer
R3 = protocol status flags
R4 = pointer to protocol specific information

Use

Use this call in your **terminal emulator** to request that a protocol module forwards any pending data. You should do so in either of these cases:

- if bit 0 ('data pending') of the link's poll word is set
- if the 'data pending' bit (commonly bit 2) of the protocol status flags (commonly in R3) is set on return from a Protocol... SWI call.

The **protocol module** forwards as much of the pending data as it is able to place in the emulator's receive buffer.

The protocol specific information it returns may be used for error messages. The protocol status flags it returns have the following meanings:

Bits	Value	Meaning
0-1	00	no connection opened
	01	connection pending
	10	connection open
	11	connection failed
2	0	no data pending
	1	more data pending
3	0	data is in bytes
	1	data is in words

All other bits are reserved and must be zero.

When this call returns to your **terminal emulator** you must examine the state of these flags:

- If the connection failed (bits 0 and 1 are set) and no data is pending (bit 2 is clear) you must attempt to close the connection by calling Protocol_CloseConnection.
- If the connection is pending you have made an error in your programming by trying to use the connection before it has been properly opened.
- Bit 2 ('data pending') has exactly the same meaning as bit 0 of a logical link's poll word, and is provided to reduce the amount of polling that needs to be done. If it is set you should continue the data transfer by calling Protocol_DataRequest.
- If the data is in words, each word contains one character in the least significant byte.

Related SWIs

Protocol_GetFileInfo (page 4-542), Protocol_GetFileData (page 4-543),
Protocol_GetFile (page 4-546)

Protocol_MenuItemSelected (Offset 7)

Requests that a protocol module services a menu selection

On entry

R0 = protocol module's link handle
R1 = pointer to menu selection block
R2 = x coordinate of mouse
R3 = y coordinate of mouse
R4 = emulator menu flags

On exit

R0 - R4 preserved

Use

Use this call in your **terminal emulator** to request that a protocol module services a selection made within its own menu. You should call this if you:

- get notice of a mouse click within the protocol's menu, via a Menu_Selection reason code from Wimp_Poll
- get notice of the pointer moving over a right arrow to activate one of the protocol's submenus, via a MenuWarning message

(See the descriptions of Wimp_Poll on page 3-112 and Wimp_SendMessage on page 3-193 for more details.)

The menu selection block contains:

R1	item in protocol menu that was selected (starting with 1)
R1+1	item in first protocol submenu that was selected
R1+2	item in second protocol submenu that was selected
...	
	terminated by 0 byte

Note: There are several important differences between this menu selection block and that returned by Wimp_Poll with a Menu_Selection reason code:

Wimp menu selection block

Menu items start from 0
 Each number is a word
 List is terminated by -1
 R1 gives item in main menu

Protocol menu selection block

Menu items start from 1
 Each number is a byte
 List is terminated by 0
 R1 gives item at root of protocol menu

The emulator menu flags show why you have made this call:

Bit	Value	Meaning
0	0	called because of a mouse click
	1	called because of a MenuWarning message

All other bits are reserved and must be zero.

The **protocol module** services the menu selection, either doing what the user clicked over, or displaying the necessary submenu.

Related SWIs

Protocol_GetProtocolMenu (page 4-521), Protocol_UnknownEvent (page 4-532)

Protocol_UnknownEvent (Offset 8)

Passes on Wimp events to a protocol module

On entry

R0 = pointer to Wimp event block (as returned by Wimp_Poll)

On exit

R0 preserved

Use

Use this call in your **terminal emulator** to pass on Wimp events you can't deal with to the protocol module you're using. You should also pass on idle events if the protocol module's Wimp_Poll mask (see Protocol_OpenLogicalLink) doesn't mask them out – even if your terminal emulator uses them.

The **protocol module** processes the Wimp event if it is one in which it is interested.

Related SWIs

Protocol_GetProtocolMenu (page 4-521), Protocol_MenuItemSelected (page 4-530)

Protocol_GetLinkState

(Offset 9)

Gets the state of a logical link

On entry

R0 = protocol module's link handle

On exit

R0 = terminal emulator's link handle

R1 = pointer to connection name (null-terminated)

R2 = pointer to protocol specific information, or 0

R3 = protocol status flags

Use

Use this call in your **terminal emulator** to get the state of a logical link.

One time you should do so is if an attempt you've made to open a connection has resulted in a pending connection. You should then wait for bit 0 of the logical link's poll word ('data pending') to be set before making this call to find if the connection was opened, or if it failed.

The **protocol module** returns a connection name suitable for the terminal emulator to use as a window title (if the connection is open or pending). The protocol specific information it returns may be used for error messages. The protocol status flags it returns have the following meanings:

Bits	Value	Meaning
0-1	00	no connection opened
	01	connection pending
	10	connection open
	11	connection failed
2	0	no data pending
	1	data pending

All other bits are reserved and must be zero.

When this call returns to your **terminal emulator** you must examine the state of these flags:

- If the connection failed (bits 0 and 1 are set) and no data is pending (bit 2 is clear) you must attempt to close the connection by calling Protocol_CloseConnection.

- If the connection is pending you must wait until bit 0 of the logical link's poll word is set. Then you should call either `Protocol_DataRequest` or `Protocol_GetLinkState` to find if the connection was opened, or if it failed.
- Bit 2 ('data pending') has exactly the same meaning as bit 0 of a logical link's poll word, and is provided to reduce the amount of polling that needs to be done. If it is set you should initiate the data transfer by calling `Protocol_DataRequest`.

Related SWIs

`Protocol_OpenLogicalLink` (page 4-518), `Protocol_CloseLogicalLink` (page 4-520),
`Protocol_OpenConnection` (page 4-522), `Protocol_CloseConnection` (page 4-525)

Protocol_Break

(Offset 10)

Forces a protocol module to generate a Break

On entry

R0 = protocol module's link handle

On exit

R0 = terminal emulator's link handle

R3 = protocol status flags

Use

Use this call in your **terminal emulator** to force a protocol module to generate a Break.

The **protocol module** generates a Break. The precise interpretation of this varies from module to module.

The documentation of a protocol module **must** state how it interprets this call.

The protocol status flags it returns have the following meanings:

Bits	Value	Meaning
0-1	00	no connection opened
	01	connection pending
	10	connection open
	11	connection failed
2	0	no data pending
	1	data pending

All other bits are reserved and must be zero.

When this call returns to your **terminal emulator** you must examine the state of these flags:

- If the connection failed (bits 0 and 1 are set) and no data is pending (bit 2 is clear) you must attempt to close the connection by calling Protocol_CloseConnection.
- If the connection is pending you have made an error in your programming by trying to use the connection before it has been properly opened.

- Bit 2 ('data pending') has exactly the same meaning as bit 0 of a logical link's poll word, and is provided to reduce the amount of polling that needs to be done. If it is set you should initiate the data transfer by calling Protocol_DataRequest.

Related SWIs

None

Protocol_SendFile (Offset 11)

Initiates sending a file over a protocol module's connection

On entry

R0 = protocol module's link handle
 R1 = RISC OS file type
 R2 = pointer to file name (null terminated)
 R3 = estimated size of file (in bytes)
 R4 = emulator send flags

On exit

R0 = terminal emulator's link handle
 R1 = protocol status flags

Use

Use this call in your **terminal emulator** to initiate sending a file over a protocol module's connection.

The emulator send flags have the following meanings:

Bit	Meaning when set
0	transfer cannot be safely paused (ie is a RAM transfer)
1	transfer is part of a multiple file transfer

All other bits are reserved and must be zero.

The **protocol module** must ready itself to accept the file over the terminal emulator's logical link, and to send it over the connection that is associated with the link. When it is ready it must show this by setting bit 2 of the link's poll word.

If bit 1 of the emulator send flags is set (a multiple file transfer) and the protocol module uses dialogue box(es) to show the state of the transfer, it must use the same box(es) for each file in turn, rather than using a new one for each file.

The protocol status flags it returns have the following meanings:

Bits	Value	Meaning
0-1	00	no connection opened
	01	connection pending
	10	connection open
	11	connection failed

All other bits are reserved and must be zero.

When this call returns to your **terminal emulator** you must examine the state of these flags:

- If the connection failed (bits 0 and 1 are set) and no data is pending (bit 2 of the link's poll word is clear) you must attempt to close the connection by calling Protocol_CloseConnection.
- If the connection is pending you have made an error in your programming by trying to use the connection before it has been properly opened.

When you start a file transfer with this call the link is in a paused state. You should wait for bit 2 of the link's poll word to be set before you try to resume the transfer by calling Protocol_SendFileData (see the next page).

Related SWIs

Protocol_TransmitData (page 4-526), Protocol_SendFileData (page 4-539), Protocol_AbortTransfer (page 4-541), Protocol_DirOp (page 4-547)

Protocol_SendFileData

(Offset 12)

Sends the data in a file over a protocol module's connection

On entry

R0 = protocol module's link handle
 R1 = pointer to transmit buffer
 R2 = length of transmit buffer (in bytes)
 R3 = emulator send data flags

On exit

R0 = terminal emulator's link handle
 R1 = protocol status flags

Use

Use this call in your **terminal emulator** to send the data in a file over a protocol module's connection. You can (if necessary) split the file into separate data packets and repeatedly use this call to transmit each packet.

The emulator send data flags have the following meanings:

Bit	Meaning when set
0	last data packet of a file (ie EOF)
1	no data is included – end of file transfer

All other bits are reserved and must be zero.

You must not set both these bits at once, so a file transfer must end with two calls of this SWI: the first with bit 0 set (EOF), the second with bit 1 set (end of file transfer).

The **protocol module** sends the file over the connection that is associated with the link. If it has to pause the transfer it must show when it is ready to resume by setting bit 2 of the link's poll word.

The protocol status flags it returns have the following meanings:

Bits	Value	Meaning
0-1	00	no connection opened
	01	connection pending
	10	connection open
	11	connection failed
2-3	00	transfer not started
	01	transfer paused
	10	transfer completed
	11	transfer failed or aborted

All other bits are reserved and must be zero.

When this call returns to your **terminal emulator** you must examine the state of these flags:

- If the connection failed (bits 0 and 1 are set) and the transfer is not paused (bits 2-3 do not have the value 01) you must attempt to close the connection by calling Protocol_CloseConnection.
- If the connection is pending you have made an error in your programming by trying to use the connection before it has been properly opened.
- If the transfer is paused (bits 2-3 have the value 01) you must wait for bit 2 of the link's poll word to be set before making this call again to continue the transfer.

Related SWIs

Protocol_TransmitData (page 4-526), Protocol_SendFile (page 4-537),
Protocol_AbortTransfer (page 4-541), Protocol_DirOp (page 4-547)

Protocol_AbortTransfer (Offset 13)

Aborts a file transfer

On entry

R0 = protocol module's link handle

On exit

R0 preserved

Use

Use this call in your **terminal emulator** to abort a file transfer.

The **protocol module** aborts the transfer and makes sure that the connection associated with the link is ready for other use.

Related SWIs

Protocol_SendFile (page 4-537), Protocol_SendFileData (page 4-539),
Protocol_GetFileInfo (page 4-542), Protocol_GetFileData (page 4-543),
Protocol_GetFile (page 4-546)

Protocol_GetFileInfo (Offset 14)

Requests that a protocol module initiates forwarding a pending file

On entry

R0 = protocol module's link handle

On exit

R0 = terminal emulator's link handle

R1 = RISC OS file type

R2 = pointer to file name (null terminated)

R3 = 0, or estimated size of file if available (in bytes)

Use

Use this call in your **terminal emulator** to request that a protocol module initiates forwarding a pending file. You should do so:

- if bit 1 ('file pending') of the link's poll word is set.
This will usually be as a result of your calling Protocol_GetFile to request that the file be sent.

The **protocol module** returns details of the file to the terminal emulator.

When this call returns to your **terminal emulator** you must use these details to get ready to receive the file, before calling Protocol_GetFileData to actually get the data.

Related SWIs

Protocol_DataRequest (page 4-528), Protocol_AbortTransfer (page 4-541),
Protocol_GetFileData (page 4-543), Protocol_GetFile (page 4-546),
Protocol_DirOp (page 4-547)

Protocol_GetFileData

(Offset 15)

Requests that a protocol module forwards the data in a file

On entry

R0 = protocol module's link handle
 R1 = pointer to receive buffer
 R2 = length of receive buffer (in bytes)

On exit

R0 = terminal emulator's link handle
 R1 preserved
 R2 = number of bytes of data placed in receive buffer
 R3 = protocol status flags

Use

Use this call in your **terminal emulator** to request that a protocol module forwards the data in a file.

The **protocol module** must forward the file data to the terminal emulator. It can (if necessary) split the file into separate data packets, pausing the transfer after each packet. If so, it must show when it is ready to forward the next packet by setting bit 2 of the link's poll word.

The protocol status flags it returns have the following meanings:

Bits	Value	Meaning
0-1	00	no connection opened
	01	connection pending
	10	connection open
	11	connection failed
2-3	00	transfer not started
	01	transfer paused
	10	transfer completed
	11	transfer failed or aborted

All other bits are reserved and must be zero.

When this call returns to your **terminal emulator** you must examine the state of these flags:

- If the connection failed (bits 0 and 1 are set) and the transfer is not paused (bits 2-3 do not have the value 01) you must attempt to close the connection by calling `Protocol_CloseConnection`.
- If the connection is pending you have made an error in your programming by trying to use the connection before it has been properly opened.
- If the transfer is paused (bits 2-3 have the value 01) you must wait for bit 2 of the link's poll word to be set before making this call again to continue the transfer.

Related SWIs

`Protocol_DataRequest` (page 4-528), `Protocol_AbortTransfer` (page 4-541),
`Protocol_GetFileInfo` (page 4-542), `Protocol_GetFile` (page 4-546),
`Protocol_DirOp` (page 4-547)

Protocol_MenuHelp (Offset 16)

Requests that a protocol module sends its interactive help message for a menu entry

On entry

R0 = protocol module's link handle

R1 = pointer to menu selection array, relative to protocol-specific menu tree

On exit

R0, R1 preserved

Use

Use this call in your **terminal emulator** to request that a protocol module sends its interactive help message for the menu entry. The menu selection array you send must be terminated by a null.

The **protocol module** must send the appropriate help message.

Related SWIs

Protocol_GetProtocolMenu (page 4-521), Protocol_MenuItemSelected (page 4-530)

Protocol_GetFile (Offset 17)

Requests that a protocol module gets a file over a connection

On entry

R0 = protocol module's link handle

R1 = pointer to file name (null terminated)

On exit

R0, R1 preserved

Use

Use this call in your **terminal emulator** to request that a protocol module gets a file over a connection.

The **protocol module** gets the necessary information to respond to a Protocol_GetFileInfo call, and the first packet of the file to respond to a Protocol_GetFileData call, before showing that it is ready by setting bit 1 ('file pending') of the link's poll word.

Related SWIs

Protocol_DataRequest (page 4-528), Protocol_AbortTransfer (page 4-541),
Protocol_GetFileInfo (page 4-542), Protocol_GetFileData (page 4-543),
Protocol_DirOp (page 4-547)

Protocol_DirOp

(Offset 18)

Performs various directory operations over a connection

On entry

R0 = protocol module's link handle

R1 = reason code

R2 = pointer to directory name – reason codes 1 & 2 only (null terminated)

On exit

R0 = terminal emulator's link handle

R1, R2 preserved

R3 = protocol status flags

Use

Use this call in your **terminal emulator** to perform various directory operations over a connection. The type of operation is set by a reason code in R1:

Reason code	Type of operation
0	null – see below
1	create directory
2	move into directory
3	move up one level in directory tree

The **protocol module** performs the specified operation. The protocol status flags it returns have the following meanings:

Bits	Value	Meaning
0-1	00	no connection opened
	01	connection pending
	10	connection open
	11	connection failed
2-3	00	invalid context
	01	operation in progress – paused
	10	operation completed
	11	operation failed or aborted

All other bits are reserved and must be zero.

When this call returns to your **terminal emulator** you must examine the state of these flags:

- If the connection failed (bits 0 and 1 are set) and there is no operation in progress (bits 2-3 do not have the value 01) you must attempt to close the connection by calling `Protocol_CloseConnection`.
- If the connection is pending you have made an error in your programming by trying to use the connection before it has been properly opened.
- If the operation is still in progress (bits 2-3 have the value 01) you must wait for bit 2 of the link's poll word to be set. You can then make this call again with a null reason code to read the flags for the completed operation.

Related SWIs

`Protocol_SendFile` (page 4-537), `Protocol_SendFileData` (page 4-539),
`Protocol_AbortTransfer` (page 4-541), `Protocol_GetFileInfo` (page 4-542),
`Protocol_GetFileData` (page 4-543), `Protocol_GetFile` (page 4-546)

Introduction

Various names and numbers that appear in RISC OS must be registered with Acorn to ensure that they don't clash with those used by other programmers. This appendix tells you what those names and numbers are, and how to register them with Acorn.

Generally, you can propose the name(s) that you would like to use, and will be allocated them if they are previously unused. However, numbers are normally allocated consecutively, so you are unlikely to have any choice as to which ones you are allocated.

Acorn keeps a single central set of header files that record all such names and numbers. Your request will be checked against the relevant file. Finally, your allocation will be recorded in the file, and you will be informed of it.

Things requiring registration

Filetypes

If you need to use a new filetype, you must register it with Acorn.

You should give a proposed textual equivalent for the filetype (8 characters maximum, as used by the 'Full info' Filer displays), and a more complete description of the filetype's functionality and/or conformance to any standards. Acorn will then inform you whether your name is unique, and – if it is unique – which filetype number you have been allocated.

For a list of currently defined filetypes, see *Table C: File types* on page 4-563.

Associated sprites

Registering filetypes is necessary to prevent any clashes in the Wimp's sprite pool between different 'file_XXX' and 'small_XXX' sprites (where XXX is a hexadecimal filetype) used by the Filer to display the filetype. Once you have registered a filetype, you may consider such sprites as also registered.

Associated system variables

Registering filetypes is also necessary to prevent any clashes between `File$Type_XXX`, `Alias$@LoadType_XXX`, `Alias$@PrintType_XXX` and `Alias$@RunType_XXX` system variables (where `XXX` is a hexadecimal filetype). Once you have registered a filetype, you may consider such variables as also registered.

SWI chunk numbers and names

If you need to supply your own SWIs, you must ask Acorn for an allocation of a SWI chunk number, the use of the SWIs within which you can then determine yourself.

You should give a proposed name for the SWI chunk. Acorn will then inform you whether your name is unique, and – if it is unique – which SWI chunk number you have been allocated.

SWIs are named as *ChunkName_FunctionName* (so in `Wimp_Initialise`, `Wimp` is the chunk name, and `Initialise` is the function name). The chunk name is normally the name of the application or module providing the SWI, which will itself need registration – see below.

For more information on SWI numbers and names, see the chapter entitled *An introduction to SWIs* on page 1-23.

Wimp message numbers

Wimp message numbers are allocated by Acorn from the same number space as SWI numbers. If you need to use a new Wimp message and have a SWI chunk allocated, you may use as Wimp message numbers the same 64 numbers that are held in that SWI chunk. Otherwise you must ask Acorn for an allocation of a range of Wimp message numbers, the use of which you can then determine yourself.

For more information on Wimp messages, see *Wimp_SendMessage (SWI &400E7)* on page 3-193.

Error numbers

If you need to generate your own errors, you must ask Acorn for an allocation of a range of error numbers, the use of which you can then determine yourself.

For more information on error numbers, see the section entitled *Error numbers* on page 1-42.

Filing system numbers and names

If you create your own filing system, you must register it with Acorn.

You should give a proposed name for the filing system, and a more complete description of its functionality and/or conformance to any standards. Acorn will then inform you whether your name is unique, and – if it is unique – which filing system number you have been allocated.

For a list of currently defined filing system numbers, see the section entitled *Filing system information word* on page 2-532.

Expansion cards: manufacturer codes and product type codes

If you create an expansion card, you must ask Acorn for an allocation of a manufacturer code and a product type code.

You should give a brief description of its functionality and/or conformance to any standards. Acorn will then inform you which codes you have been allocated.

For more information on these codes, see the section entitled *Extended Expansion Card Identity* on page 4-122.

CMOS RAM bytes

There are 4 bytes of CMOS RAM reserved for each expansion card slot, which your expansion cards may freely use; see the section entitled *Non-volatile memory (CMOS RAM)* on page 1-363. For all other purposes you should remember state in some other manner (for example using an *App\$Options* system variable in a desktop boot file, or using a Choices file within your application). It is only in very exceptional circumstances that Acorn may allocate CMOS RAM bytes to other parties.

Territory, country and alphabet numbers and names

If you need to use a new territory, country, or alphabet, you must register it with Acorn.

You should give a proposed name for the territory, country, or alphabet, and (for alphabets) a more complete description of its functionality and/or conformance to any standards. Acorn will then inform you whether your name is unique, and – if it is unique – which territory, country, or alphabet number you have been allocated.

For a list of currently defined country and alphabet numbers, see the section entitled *Names and numbers* on page 3-768.

DrawFile object types and tagged object types

If you need to use a new object type or tagged object type in a Draw file, you must register it with Acorn.

For an object type you should give full details of its file format. For a tagged object type you should give a brief description of the purpose of the tag. Acorn will then inform you which type numbers you have been allocated.

For a list of currently defined object types and tagged object types, see the section entitled *Draw files* on page 4-461.

Module names

If you create a new module, you must register it with Acorn, since only one module of a given name can be loaded at once.

You should give a proposed name for the module and a brief description of its functionality. Acorn will then inform you whether your name is unique, and hence if you may use it.

Associated system variables

Registering module names is also necessary to prevent any clashes between system variables used by modules, such as *Module\$Options*. Once you have registered the module name '*Module*', you may consider all variables beginning with '*Module\$*' as also registered.

To ensure there are no clashes with '*App\$*' or '*Resource\$*' system variables, Acorn will also check that your module name does not match any **other** programmers' registered application or shared resource names. However, you may register identical module, application and /or shared resource names; it is then your responsibility to prevent any clashes between your **own** system variables.

Application names

If you create a new application, you must register it with Acorn.

You should give a proposed name for the application and a brief description of its functionality. Acorn will then inform you whether your name is unique, and hence if you may use it.

Associated sprites

Registering application names is necessary to prevent any clashes in the Wimp's sprite pool between different application's '*!app*' and '*sm!app*' sprites, used by the Filer to display the application directory's icon. Once you have registered an application name, you may consider such sprites as also registered.

Associated system variables

Registering application names is also necessary to prevent any clashes between system variables used by applications, such as *App\$Dir* or *App\$Options*. Once you have registered the application name ‘*App*’, you may consider all variables beginning with ‘*App\$*’ as also registered.

To ensure there are no clashes with ‘*Module\$*’ or ‘*Resource\$*’ system variables, Acorn will also check that your application name does not match any **other** programmers’ registered module or shared resource names. However, you may register identical module, application and /or shared resource names; it is then your responsibility to prevent any clashes between your **own** system variables.

Shared resources

If you create a new shared resource directory, you must register it with Acorn.

You should give a proposed name for the shared resource and a brief description of its functionality. Acorn will then inform you whether your name is unique, and hence if you may use it.

Associated sprites

Registering shared resource names is necessary to prevent any clashes in the Wimp’s sprite pool between different shared resource’s ‘!*resource*’ and ‘sm!*resource*’ sprites (used by the Filer to display the shared resource directory’s icon). Once you have registered an shared resource name, you may consider such sprites as also registered.

Associated system variables

Registering shared resource names is also necessary to prevent any clashes between system variables used by shared resources, such as *Resource\$Dir*. Once you have registered the shared resource name ‘*Resource*’, you may consider all variables beginning with ‘*Resource\$*’ as also registered.

To ensure there are no clashes with ‘*Module\$*’ or ‘*App\$*’ system variables, Acorn will also check that your shared resource name does not match any **other** programmers’ registered module or application names. However, you may register identical module, application and /or shared resource names; it is then your responsibility to prevent any clashes between your **own** system variables.

* Commands

If you create a new * Command, you must register it with Acorn.

You should give a proposed name for the command, and a brief description of its functionality. Acorn will then inform you whether your name is unique, and hence if you may use it.

Sprite names

If you add a sprite to the Wimp sprite pool – for example using *IconSprites – you must register it with Acorn.

You should give a proposed name for the sprite. Acorn will then inform you whether your name is unique, and hence if you may use it.

Provided you have registered a filetype, application or shared resource, you need not register the associated sprites that the Filer uses to display them. See page 4-549, page 4-552 and page 4-553 respectively.

You should not register the names of sprites that are held in your applications' own sprite areas. Desktop applications must not use the system sprite pool.

Font names

If you create a new font, you must register it with Acorn.

You should give a proposed name for the font. Acorn will then inform you whether your name is unique, and hence if you may use it.

Device numbers

If you need to add a new device, you must ask Acorn for an allocation of a major and a minor device number.

You should give a brief description of the device's functionality. Acorn will then inform you which device numbers you have been allocated.

Printer driver and printer dumper numbers

If you create a new printer driver or dumper module, you must ask Acorn for an allocation of a printer driver or dumper number.

You should give a brief description of the printer driver or dumper's functionality. Acorn will then inform you which ID number you have been allocated.

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	Page
0	@	0	Does nothing	1-570
1	A	1	Sends next character to printer only	1-571
2	B	0	Enables printer	1-572
3	C	0	Disables printer	1-573
4	D	0	Writes text at text cursor	1-574
5	E	0	Writes text at graphics cursor	1-575
6	F	0	Enables VDU driver	1-576
7	G	0	Generates bell sound	1-577
8	H	0	Moves cursor back one character	1-578
9	I	0	Moves cursor on one space	1-579
10	J	0	Moves cursor down one line	1-580
11	K	0	Moves cursor up one line	1-581
12	L	0	Clears text window	1-582
13	M	0	Moves cursor to start of current line	1-583
14	N	0	Turns on paged mode	1-584
15	O	0	Turns off paged mode	1-585
16	P	0	Clears graphics window	1-586
17	Q	1	Defines text colour	1-587
18	R	2	Defines graphics colour	1-588
19	S	5	Defines logical colour	1-590
20	T	0	Restores default logical colours	1-594
21	U	0	Disables VDU drivers	1-595
22	V	1	Selects screen mode	1-596

VDU code	Ctrl plus	Extra bytes	Meaning	Page
23	W	9	Multi-purpose command:	1-601
23,0			Sets the interlace and controls cursor appearance	1-602
23,1			Controls text cursor appearance	1-603
23,2-5			Defines ECF pattern and colours	1-604
23,6			Sets dot-dash line style	1-605
23,7			Scrolls text window or screen	1-606
23,8			Clears a block of the text window	1-608
23,9			Sets first flash time	1-610
23,10			Sets second flash time	1-611
23,11			Sets default patterns	1-612
23,12-15			Defines simple ECF patterns and colours	1-614
23,16			Controls cursor movement after printing	1-616
23,17,0-3			Sets the tint for a colour	1-618
23,17,4			Chooses ECF patterns	1-619
23,17,5			Exchanges text foreground and background colours	1-620
23,17,6			Sets ECF origin	1-621
23,17,7			Sets character size/spacing	1-622
23,18-24			Reserved for future expansion	1-623
23,25-26			Private Font Manager calls	1-624
23,27			Private Sprite Manager calls	1-625
23,28-31			Reserved for use by application programs	1-626
23,32-255			Redefines printable characters	1-627
24	X	8	Defines graphics window	1-629
25	Y	5	PLOT command	1-630
26	Z	0	Restores default windows	1-633
27	[0	Does nothing	1-634
28	\	4	Defines text window	1-635
29]	4	Defines graphics origin	1-636
30	^	0	Homes text cursor	1-637

VDU code	Ctrl plus	Extra bytes	Meaning	Page
31	–	2	Moves text cursor	1-638
127		0	Delete	1-639

The modes available in RISC OS depend on the configured monitor type (see *Configure MonitorType on page 1-761) and the model of computer. Below is a table of all modes provided by RISC OS, which shows:

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

Type	Monitor
0	50Hz TV standard colour or monochrome monitor
1	Multi-frequency monitor
2	64Hz high-resolution monochrome monitor
3	60Hz VGA-type monitor
4	Super-VGA-type monitor (not available in RISC OS 2)
5	LCD (liquid crystal display) (not available in RISC OS 2)
- the notes on the following page that are relevant to the mode.

Mode	Text resolution	Pixel resolution	OS units resolution	Logical colours	Mem used	Refresh rate	Band-width	Monitor types	Notes
0	80 × 32	640 × 256	1280 × 1024	2	20K	50Hz	1M/s	0,1,3,4,5	↯
1	40 × 32	320 × 256	1280 × 1024	4	20K	50Hz	1M/s	0,1,3,4,5	↯
2	20 × 32	160 × 256	1280 × 1024	16	40K	50Hz	2M/s	0,1,3,4,5	↯
3	80 × 25	Text only	Text only	2	40K	50Hz	2M/s	0,1,3,4,5	↯fý
4	40 × 32	320 × 256	1280 × 1024	2	20K	50Hz	1M/s	0,1,3,4,5	↯
5	20 × 32	160 × 256	1280 × 1024	4	20K	50Hz	1M/s	0,1,3,4,5	↯
6	40 × 25	Text only	Text only	2	20K	50Hz	1M/s	0,1,3,4,5	↯fý
7	40 × 25	Teletext	Teletext	16	80K	50Hz	4M/s	0,1,3,4,5	↯f
8	80 × 32	640 × 256	1280 × 1024	4	40K	50Hz	2M/s	0,1,3,4,5	↯
9	40 × 32	320 × 256	1280 × 1024	16	40K	50Hz	2M/s	0,1,3,4,5	↯
10	20 × 32	160 × 256	1280 × 1024	256	80K	50Hz	4M/s	0,1,3,4,5	↯
11	80 × 25	640 × 250	1280 × 1000	4	40K	50Hz	2M/s	0,1,3,4,5	↯«
12	80 × 32	640 × 256	1280 × 1024	16	80K	50Hz	4M/s	0,1,3,4,5	↯
13	40 × 32	320 × 256	1280 × 1024	256	80K	50Hz	4M/s	0,1,3,4,5	↯
14	80 × 25	640 × 250	1280 × 1000	16	80K	50Hz	3.9M/s	0,1,3,4,5	↯«
15	80 × 32	640 × 256	1280 × 1024	256	160K	50Hz	8M/s	0,1,3,4,5	↯
16	132 × 32	1056 × 256	2112 × 1024	16	132K	50Hz	6.6M/s	0,1	Ý
17	132 × 25	1056 × 250	2112 × 1000	16	132K	50Hz	6.5M/s	0,1	Ý«
18	80 × 64	640 × 512	1280 × 1024	2	40K	50Hz	2M/s	1	
19	80 × 64	640 × 512	1280 × 1024	4	80K	50Hz	4M/s	1	
20	80 × 64	640 × 512	1280 × 1024	16	160K	50Hz	8M/s	1	
21	80 × 64	640 × 512	1280 × 1024	256	320K	50Hz	16M/s	1	
22	96 × 36	768 × 288	768 × 576	16	108K	50Hz	5.4M/s	0,1	¿¥
23	144 × 56	1152 × 896	2304 × 1792	2	126K	64Hz	8.1M/s	2	
24	132 × 32	1056 × 256	2112 × 1024	256	264K	50Hz	13.2M/s	0,1	Ý
25	80 × 60	640 × 480	1280 × 960	2	37.5K	60Hz	2.3M/s	1,3,4,5	
26	80 × 60	640 × 480	1280 × 960	4	75K	60Hz	4.5M/s	1,3,4,5	
27	80 × 60	640 × 480	1280 × 960	16	150K	60Hz	9M/s	1,3,4,5	
28	80 × 60	640 × 480	1280 × 960	256	300K	60Hz	18M/s	1,3,4,5	
29	100 × 75	800 × 600	1600 × 1200	2	58.6K	56Hz	3.3M/s	1,4	¿i
30	100 × 75	800 × 600	1600 × 1200	4	117.2K	56Hz	6.6M/s	1,4	¿i
31	100 × 75	800 × 600	1600 × 1200	16	234.4K	56Hz	13.2M/s	1,4	¿i
33	96 × 36	768 × 288	1536 × 1152	2	27K	50Hz	1.4M/s	0,1	¿
34	96 × 36	768 × 288	1536 × 1152	4	54K	50Hz	2.7M/s	0,1	¿
35	96 × 36	768 × 288	1536 × 1152	16	108K	50Hz	5.4M/s	0,1	¿
36	96 × 36	768 × 288	1536 × 1152	256	216K	50Hz	10.8M/s	0,1	¿
37	112 × 44	896 × 352	1792 × 1408	2	38.5K	60Hz	2.3M/s	1	¿
38	112 × 44	896 × 352	1792 × 1408	4	77K	60Hz	4.6M/s	1	¿
39	112 × 44	896 × 352	1792 × 1408	16	154K	60Hz	9.2M/s	1	¿
40	112 × 44	896 × 352	1792 × 1408	256	308K	60Hz	18.5M/s	1	¿
41	80 × 44	640 × 352	1280 × 1408	2	27.5K	60Hz	1.7M/s	1,3,4,5	¿→Đ
42	80 × 44	640 × 352	1280 × 1408	4	55K	60Hz	3.3M/s	1,3,4,5	¿→Đ
43	80 × 44	640 × 352	1280 × 1408	16	110K	60Hz	6.6M/s	1,3,4,5	¿→Đ
44	80 × 25	640 × 200	1280 × 800	2	15.7K	60Hz	0.9M/s	1,3,4,5	¿↯
45	80 × 25	640 × 200	1280 × 800	4	31.3K	60Hz	1.9M/s	1,3,4,5	¿↯
46	80 × 25	640 × 200	1280 × 800	16	62.5K	60Hz	3.8M/s	1,3,4,5	¿↯

Notes on display modes

- 1 These modes are not available in RISC OS 2.00, nor (except for mode 31) are they available in RISC OS 2.01.
- 2 These modes are not available on early models of RISC OS computers (ie the Archimedes 300, 400 and 400/1 series, and the A3000), because they are unable to clock VIDC at the necessary rate.
- 3 These modes are handled differently with a VGA or Super-VGA-type monitor. **If you are using such a monitor:**
 - RISC OS 2.00 does not implement these modes.
 - These modes are all displayed on a screen having 352 raster lines. Where a mode has fewer than 352 vertical pixels, it is centred on the screen with blank rasters at the top and bottom. Because of their appearance these modes are known as *letterbox modes*.
 - The refresh rate is 70Hz.
 - The bandwidths shown in the table for these modes are lower than these monitor types consume, because no allowance has been made for the blank rasters.
 - Early models of RISC OS computers (ie the Archimedes 300, 400 and 400/1 series, and the A3000) scan these modes some 4.7% slow. Again this is because they are unable to clock VIDC at the necessary rate. Most VGA and Super-VGA-type monitors can still successfully lock onto this signal, but some may not. Furthermore, these models do not provide a *Sync Polarity* signal. This makes the effect of letterbox modes (see above) more severe.
- 4 Early models of RISC OS computers (ie the Archimedes 300, 400 and 400/1 series, and the A3000) also scan these modes some 4.7% slow with multi-frequency monitors. Again this is because they are unable to clock VIDC at the necessary rate.
- 5 These modes do not display graphics, and are provided for compatibility with BBC/Master series computers.
- 6 In these modes circles, arcs, sectors and segments do not look circular. This is because the aspect ratio of the pixels is not in a 1:2, 1:1 or 2:1 ratio.
- 7 These are *gap modes*, where the colour of the gaps is not necessarily the same as the text background.
- 8 These modes are not a multiple of eight pixels high. By default, in these modes the bottom of the screen corresponds to the bottom line of ECF patterns, but the top line will not correspond to the top line of ECF patterns.
- 9 This mode is not available in RISC OS 3 (version 3.00). It provides a double-sized display suitable for use by visually impaired people. Unfortunately some applications may not provide correct displays when used with this mode.

Other notes

Mode 32 has not been defined.

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

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

List of file types

File types are three-digit hexadecimal numbers. They are divided into ranges:

E00 - FFF	allocated by Acorn for generic data types
B00 - DFF	allocated by Acorn to software houses for applications
A00 - AFF	reserved for use by Acorn applications
400 - 9FF	allocated by Acorn to software houses for applications
100 - 3FF	allocated by Acorn to public domain applications
000 - 0FF	free for users

For information about the allocation of file types, see *Appendix H: Registering names* on page 4-549.

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 start-up, 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 double dagger ‘‡’, or by a single dagger ‘†’ if not available in RISC OS 2. For example, file type &FFF is set at start-up to have the textual equivalent *Text*. Other textual equivalents may be set as an application is first ‘seen’ by the Filer, or as it 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 hexadecimal file type.

You should use the hexadecimal file type in command scripts and in programs, otherwise you will find that your files will give an error if you try to run them on a machine that uses a territory with different textual equivalents.

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:

Acorn file types

Type	Description	Textual equivalent	
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 operations)	BBC font	‡
FF6	Font (4 bpp bitmap only)	Font	‡
FF5	PostScript	PoScript	‡
FF4	Dot Matrix data file	Printout	†
FF3	LaserJet data file	LaserJet	
FF2	Configuration (CMOS RAM)	Config	†
FF1	Raw unprocessed data (eg terminal streams)	RawData	
FF0	Tagged Image File Format	TIFF	
FED	Palette data	Palette	‡
FEC	Template file	Template	‡
FEB	Obey file	Obey	‡
FEA	Desktop	Desktop	†
FE9	ViewWord	ViewWord	
FE8	ViewPS	ViewPS	
FE7	ViewSheet	ViewSht	
FE6	UNIX executable	UNIX Ex	
FE4	DOS file	DOS	†
FE3	Atari file	Atari	
FE2	Commodore Amiga file	Amiga	
FE1	Make data	Make	
FDF	TCP/IP suite: VT220 script	VTScript	
FDE	TCP/IP suite: VT220 setup	VTSetup	
FDD	Master utilities	MasterUtl	
FDC	TCP/IP suite: unresolvable UNIX soft link	SoftLink	
FDB	Text using CR and LF for line ends	TextCRLF	
FDA	PC Emulator: DOS batch file	MSDOSbat	
FD9	PC Emulator: DOS executable file	MSDOSexe	
FD8	PC Emulator: DOS command file	MSDOScom	
FD7	Obey file in a task window	TaskObey	†
FD6	Exec file in a task window	TaskExec	†
FD5	DOS Pict	Pict	
FD4	International MIDI Assoc. MIDIfiles standard	MIDI	

FD3	Acorn DDE: debuggable image	DebImage
FD1	BASIC stored as text	BASICTxt
FD0	PC Emulator: configuration	PCEmConf
FCF	Font cache	FontCache †
FCE	FileCore floppy disc image	FileCoreFloppyDisc
FCD	FileCore hard disc image	FileCoreHardDisc
FCC	Device object within DeviceFS	Device †
FCA	Single compressed file	Squash
FC9	Sun raster file	SunRastr
FC8	DOS MultiFS disc image	DOSDisc ‡
FC7	Macintosh format Type 1 font	MacType1
FC6	!Printers printer definition file	PrintDfn
FC3	!Patch patch definition file	Patch
FC2	Audio Interchange file format	AIFF

Industry standard file types

Type	Description	Textual equivalent
DFE	Comma separated variables	CSV
DEA	Data exchange format (AutoCAD etc)	DXF
DB4	SuperCalc III file	SuperCalc
DB3	DBase III file	DBaseIII
DB2	DBase II	DBaseII
DB1	DBase index file	DBaseIndex
DB0	Lotus 123 WK1 format	WK1
CE5	T _E X file	TeX
CB6	Amiga Sound Tracker	AmigaSTM
CAF	IGIS graphics	IGIS
CAE	Hewlett-Packard graphics language	HPGLPlot
C85	JPEG (Joint Photographic Experts Group) file	JPEG
C35	Corel Draw file	CorlDraw

BBC ROM file type

Type	Description	Textual equivalent
BBC	BBC ROM file (ROMFS)	BBC ROM ‡

Acornsoft file types

Type	Description	Textual equivalent
AFF	Draw file	DrawFile †
AFE	Mouse event record	Mouse
AFA	DTP style file	DtpStyle

List of file types

AF9	DTP documents	DtpDoc
AF8	First Word Plus file	1stWord+
AF7	Help file	HelpInfo
AF1	Maestro file	Music
AF0	ArcWriter file	ARCWriter
AE9	Alarm file	Alarms
ADB	Outline font	New Font

Introduction

This chapter includes tables of all the alphabet sets available on your Acorn computer. Most are based on the International Standards Organisation ISO 8859 document.

Loading alphabets

When you load an alphabet it overlays the previous alphabet. Most alphabets have a number of undefined characters, shown in the tables below by a light grey square. In such cases, the previous character definition for that code remains in effect.

The character codes 0 - 31 and 127 are not printable characters; they have special meaning to the VDU drivers, as described in the chapter entitled *VDU Drivers* on page 1-549. They are represented in the tables below by a dark grey square.

You can load alphabets using OS_Byte 71 (page 3-780) or *Alphabet (page 3-783).

How alphabets are initially set up

The default alphabet

When the kernel is booted it sets up a default alphabet.

The kernel's default alphabet always contains all characters that are defined in the Latin1 alphabet for the release of RISC OS in use (see page 4-569). Note that this definition has been gradually extended by the addition of extra characters in the range &80 - &9F (128 - 159).

The kernel's representation of characters that are neither defined in the Latin1 alphabet nor used by the VDU drivers varies. In RISC OS 2 they are represented by the underlined string 'These characters are not defined', and in RISC OS 3 by the hexadecimal value of their character code. In the future some of these undefined characters may be used to further extend the Latin1 alphabet, or their representation may change. Furthermore, it is these characters that users are most likely to redefine if necessary. Consequently, you must not rely upon their initial representation.

The configured alphabet

The default alphabet is then overlaid by the alphabet that is correct for the computer's configured territory, as set by *Configure Territory (page 3-854). Under RISC OS 2, the alphabet used is instead determined by the computer's configured country; see *Configure Country on page 3-786.

The window manager

When the window manager starts, it redefines some characters. In RISC OS 2 these were used to draw windows' borders, and so have to be present for the desktop to have the correct appearance. Later versions of RISC OS still redefine some of these characters for backwards compatibility, but do not themselves use them. You must not rely on the presence of these characters unless your program is running under the desktop in RISC OS 2.

Keyboard shortcuts

The description of the *Country command on page 3-789 explains the relationship between *country*, *alphabet* and *keyboard*. There are some useful keyboard shortcuts which you can use to access various characters and 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 keyboard layouts:

- | | |
|-------------|---|
| 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). |

and the other allows you to access top bit set characters without using the Chars application:

- | | |
|--|---|
| Alt <decimal character code typed on numeric keypad> | Enters the character corresponding to the character code typed. |
|--|---|

The following sequence also switches the keyboard layout:

- 1 Press and hold Alt and Ctrl together.
- 2 Press F12.
- 3 Release Ctrl.
- 4 Still holding Alt, type on the numeric keypad the international telephone dialling code for the country you want (eg 49 for Germany, 39 for Italy, 33 for France).
- 5 Release Alt.

Latin1 alphabet (ISO 8859/1)

This is the default alphabet used by Acorn computers.

	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	
+0			0	@	P	`	p		'		°	À	Ð	à	õ		0
+1			!	1	A	Q	a	q	Ŵ	'	ı	±	Á	Ñ	á	ñ	1
+2			"	2	B	R	b	r	ŵ	<	ç	²	Â	Ò	â	ò	2
+3			#	3	C	S	c	s		>	£	³	Ã	Ó	ã	ó	3
+4			\$	4	D	T	d	t		“	¤	'	Ä	Ô	ä	ô	4
+5			%	5	E	U	e	u	Û	”	¥	µ	Å	Õ	å	õ	5
+6			&	6	F	V	f	v	ÿ	„	ı	¶	Æ	Ö	æ	ö	6
+7			'	7	G	W	g	w		–	§	·	Ç	×	ç	÷	7
+8			(8	H	X	h	x		—	”	,	È	Ø	è	ø	8
+9)	9	I	Y	i	y		–	©	¹	É	Ù	é	ù	9
+10			*	:	J	Z	j	z		œ	ª	º	Ê	Ú	ê	ú	A
+11			+	;	K	[k	{		œ	«	»	Ë	Û	ë	û	B
+12			,	<	L	\	l		...	†	¬	¼	Ì	Ü	ì	ü	C
+13			-	=	M]	m	}	™	‡	-	½	Í	Ý	í	ý	D
+14			.	>	N	^	n	~	‰	fi	®	¾	Î	Þ	î	þ	E
+15			/	?	O	_	o		•	fl	ˆ	¿	Ï	ß	ï	ÿ	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

In RISC OS 2 characters &80 - &9F (128 - 159) are undefined.

In RISC OS 3 (version 3.00) characters &80 - &8B (128 - 139) are undefined.

Latin2 alphabet (ISO 8859/2)

	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240		
+0			0	@	P	`	p		'		°	Ř	Đ	í	ď		0	
+1			!	1	A	Q	a	q		'	Ą	ą	Á	Ń	á	ń		1
+2			"	2	B	R	b	r		<	˘	Ĉ	Ĉ	â	ĉ			2
+3			#	3	C	S	c	s		>	Ł	ł	Ǻ	Ó	ǻ	ó		3
+4			\$	4	D	T	d	t		"	α	'	Ä	Ô	ä	ô		4
+5			%	5	E	U	e	u		"	Ľ	ĺ	Ĺ	Ő	ĺ	ő		5
+6			&	6	F	V	f	v		"	Š	š	Ć	Ö	ć	ö		6
+7			'	7	G	W	g	w		-	§	˘	Ç	×	ç	÷		7
+8			(8	H	X	h	x		—	"	˘	Č	Ř	č	ř		8
+9)	9	I	Y	i	y		-	Š	š	É	Ú	é	ú		9
+10			*	:	J	Z	j	z		Œ	Ş	ş	Ę	Ú	ę	ú		A
+11			+	;	K	[k	{		œ	ř	ř	Ě	Ů	ě	ů		B
+12			,	<	L	\	l		...	†	Ž	ž	Ě	Ü	ě	ü		C
+13			-	=	M]	m	}	™	‡	-	"	Í	Ý	í	ý		D
+14			.	>	N	^	n	~	‰	fi	Ž	ž	Î	Ť	î	ť		E
+15			/	?	O	_	o		•	fl	Ž	ž	Ď	ß	ď	·		F
			0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

In RISC OS 2 characters &80 - &9F (128 - 159) are undefined.

Latin3 alphabet (ISO 8859/3)

	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	
+0				0	@	P	`	p		'		°	À		à		0
+1			!	1	A	Q	a	q		'	Ɔ	ħ	Á	Ñ	á	ñ	1
+2			"	2	B	R	b	r		<	˘	²	Â	Ò	â	ò	2
+3			#	3	C	S	c	s		>	£	³		Ó		ó	3
+4			\$	4	D	T	d	t		"	¤	'	Ä	Ô	ä	ô	4
+5			%	5	E	U	e	u		"		µ	Č	Ĝ	č	ĝ	5
+6			&	6	F	V	f	v		„	Ĥ	ĥ	Ĉ	Ö	ĉ	ö	6
+7			'	7	G	W	g	w		–	§	·	Ç	×	ç	÷	7
+8			(8	H	X	h	x		—	¨	‚	È	Ĝ	è	ĝ	8
+9)	9	I	Y	i	y		–	ı	ı	É	Ù	é	ù	9
+10			*	:	J	Z	j	z		œ	Ŗ	ŗ	Ê	Ú	ê	ú	A
+11			+	;	K	[k	{		œ	Ǧ	ǧ	Ë	Û	ë	û	B
+12			,	<	L	\	l		...	†	Ĵ	ĵ	Ì	Ü	ì	ü	C
+13			-	=	M]	m	}	™	‡	-	½	Í	Ŭ	í	ŭ	D
+14			.	>	N	^	n	~	‰	fi			Î	Ŝ	î	ŝ	E
+15			/	?	O	_	o		•	fl	Ž	ž	Ï	ß	ï	·	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

In RISC OS 2 characters &80 - &9F (128 - 159) are undefined.

Latin4 alphabet (ISO 8859/4)

	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	
+0			0	@	P	`	p		'		°	Ā	Đ	ā	đ		0
+1			!	1	A	Q	a	q		'	Ą	ą	Á	Ń	á	ń	1
+2			"	2	B	R	b	r		<	κ	Ĉ	Ď	ĉ	ď		2
+3			#	3	C	S	c	s		>	Ŕ	ŕ	Ã	Ț	ã	ț	3
+4			\$	4	D	T	d	t		"	α	'	Ä	Ô	ä	ô	4
+5			%	5	E	U	e	u		"	ĩ	ĩ	Å	Õ	å	õ	5
+6			&	6	F	V	f	v		"	Ł	ł	Æ	Ö	æ	ö	6
+7			'	7	G	W	g	w		-	§	˘	Į	×	į	÷	7
+8			(8	H	X	h	x		—	˝	˝	Č	Ø	č	ø	8
+9)	9	I	Y	i	y		-	Š	š	É	Ț	é	ț	9
+10			*	:	J	Z	j	z		œ	Ě	ě	Ę	Ú	ę	ú	A
+11			+	;	K	[k	{		œ	Ĝ	ĝ	Ě	Û	ë	û	B
+12			,	<	L	\	l		...	†	Ʀ	Ʀ	É	Ü	è	ü	C
+13			-	=	M]	m	}	™	‡	-	Đ	Í	Û	í	ü	D
+14			.	>	N	^	n	~	‰	fi	Ž	ž	Î	Ū	î	ū	E
+15			/	?	O	_	o		•	fl	-	ŋ	Ī	β	ī	'	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

In RISC OS 2 characters &80 - &9F (128 - 159) are undefined.

Cyrillic alphabet (ISO 8859/5)

	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	
+0			0	@	P	`	p					А	Р	а	р		0
+1			!	1	А	Q	а	q			Ё	Б	С	б	с	ё	1
+2			"	2	В	R	в	r				В	Т	в	т		2
+3			#	3	С	S	с	s				Г	У	г	у		3
+4			\$	4	Д	T	д	t				Д	Ф	д	ф		4
+5			%	5	Е	U	е	u				Е	Х	е	х		5
+6			&	6	Ф	V	ф	v				Ж	Ц	ж	ц		6
+7			'	7	Г	W	г	w				З	Ч	з	ч		7
+8			(8	Н	X	н	x				И	Ш	и	ш		8
+9)	9	І	Y	і	y				Й	Щ	й	щ		9
+10			*	:	J	Z	ј	z				К	Ъ	к	ъ		А
+11			+	;	К	[к	{				Л	Ы	л	ы		В
+12			,	<	L	\	l					М	Ь	м	ь		С
+13			-	=	M]	m	}			-	Н	Э	н	э		Д
+14			.	>	N	^	n	~				О	Ю	о	ю		Е
+15			/	?	О	_	о					П	Я	п	я		Ф
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Greek alphabet (ISO 8859/7)

	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	
+0			0	@	P	`	p				°	í	Π	ύ	π		0
+1			!	1	A	Q	a	q			´	±	A	P	α	ρ	1
+2			"	2	B	R	b	r			´	²	B		β	ς	2
+3			#	3	C	S	c	s			£	³	Γ	Σ	γ	σ	3
+4			\$	4	D	T	d	t				´	Δ	T	δ	τ	4
+5			%	5	E	U	e	u				ˆ	E	Υ	ε	υ	5
+6			&	6	F	V	f	v			ı	Á	Z	Φ	ζ	φ	6
+7			'	7	G	W	g	w			§	·	H	X	η	χ	7
+8			(8	H	X	h	x			¨	È	Θ	Ψ	θ	ψ	8
+9)	9	I	Y	i	y			©	Ĥ	I	Ω	ι	ω	9
+10			*	:	J	Z	j	z				Í	K	Ĭ	κ	ï	A
+11			+	;	K	[k	{			«	»	Λ	ÿ	λ	ÿ	B
+12			,	<	L	\	l				¬	Ó	M	ά	μ	ó	C
+13			-	=	M]	m	}			-	½	N	έ	ν	ύ	D
+14			.	>	N	^	n	~				ÿ	Ξ	ή	ξ	ώ	E
+15			/	?	O	_	o				—	Ω	O	ι	o		F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Hebrew alphabet (ISO 8859/8)

	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	
+0																	א נ 0
+1																	ב ס 1
+2																	ג ע 2
+3																	ד ף 3
+4																	ה פ 4
+5																	ו ץ 5
+6																	ז צ 6
+7																	ח ק 7
+8																	ט ר 8
+9																	י ש 9
+10																	ך ת A
+11																	כ B
+12																	ל C
+13																	ם D
+14																	מ E
+15																	ן F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Cyrillic2 alphabet (DOS code page 866)

	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	
+0			0	@	Р	`	р	А	Р	а					р		0
+1			!	1	А	Q	а	q	Б	С	б				с		1
+2			"	2	В	Р	в	р	В	Т	в				т		2
+3			#	3	С	С	с	с	Г	У	г				у		3
+4			\$	4	Д	Т	д	т	Д	Ф	д				ф		4
+5			%	5	Е	U	е	u	Е	Х	е				х		5
+6			&	6	Ф	V	ф	v	Ж	Ц	ж				ц		6
+7			'	7	Г	W	г	w	З	Ч	з				ч		7
+8			(8	Н	X	н	x	И	Ш	и				ш		8
+9)	9	І	Y	і	y	Й	Щ	й				щ		9
+10			*	:	J	Z	j	z	К	Ъ	к				ъ		А
+11			+	;	K	[k	{	Л	Ы	л				ы		В
+12			,	<	L	\	l		М	Ь	м				ь		С
+13			-	=	M]	m	}	Н	Э	н				э		Д
+14			.	>	N	^	n	~	О	Ю	о				ю		Е
+15			/	?	O	_	o		П	Я	п				я		Ф
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

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	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Nothing	Clear graphics	█	⊖	⊗	⊕	⊘	⊙	⊚	⊛	⊜	⊝	⊞	⊟	⊠	⊡
1	Next to printer	Define text colour	!	1	A	Q	a	q	ā	ā	ı	ı	A	P	α	P
2	Start printer	Define graphics colour	"	2	B	R	b	r	Ē	æ	-	˘	B	Σ	β	σ
3	Stop printer	Define logical colours	#	3	C	S	s	c	c	Γ	˘	Γ	T	γ	τ	
4	Separate cursors	Default logical colours	\$	4	D	T	d	t	É	é	-	ı	Δ	Υ	δ	υ
5	Join cursors	Disable VDU	%	5	E	U	e	u	Ö	ö	ı	ı	E	Ξ	ε	φ
6	Enable VDU	Select mode	&	6	F	V	f	v	Ü	ü	-	ı	Z	X	Z	X
7	Bell	Reprogram characters	'	7	G	W	g	w	⊖	ı	T	ı	H	Ψ	η	ψ
8	Back	Define graphics area	(8	H	X	h	x	+	ı	'	T	θ	Ω	θ	ω
9	Forward	Plot)	9	I	Y	i	y	+	ö		ı	I	V	ı	ä
A	Down	Default text / graphics areas	*	:	J	Z	j	z	↓	ö	ı	'	K	I	k	z
B	Up	Nothing	+	;	K	C	k	c	↑	ı	ı	ı	A	ı	λ	≡
C	Clear screen	Define text area	,	<	L	\	ı	ı	ä	ö	ı	ı	M	ı	μ	z
D	Start of line	Define graphics origin	-	=	M	J	m	j	è	ÿ	H	.	N	ı	v	z
E	Paged mode	Move text cursor to (0,0)	.	>	N	^	n	~	è	ÿ	H	ı	ı	ı	ı	z
F	Scroll mode	Move text cursor	/	?	O	_	o	o	Back space and delete	è	ÿ	H	ı	ı	ı	ı

Teletext characters (used only in mode 7)

Teletext alphanumeric

	0	1	2	3	4	5	6	7
0	Nothing	Nothing	■	○	Ⓔ	Ⓕ	£	Ⓗ
1	Next to printer	Nothing	!	1	A	Q	≡	g
2	Start printer	Nothing	"	2	B	R	b	r
3	Stop printer	Nothing	#	3	C	S	c	s
4	Nothing	Nothing	\$	4	O	T	d	t
5	Nothing	Disable VDU	%	5	E	U	e	u
6	Enable VDU	Select mode	&	6	F	V	f	v
7	Bell	Reprogram characters	'	7	G	W	g	w
8	Back	Nothing	<	8	H	X	h	x
9	Forward	Nothing	>	9	I	Y	i	y
A	Down	Nothing	*	:	J	Z	j	z
B	Up	Nothing	+	;	K	←	k	↳
C	Clear Screen	Nothing	,	<	L	↳	l	
D	Start of line	Nothing	-	=	M	→	m	↳
E	Paged mode	Move cursor to (0,0)	.	>	N	↑	n	÷
F	Scroll mode	Move cursor	/	?	O	-	o	Back space and delete

	8	9	A	B	C	D	E	F
0	Nothing	Nothing	█	○	@	P	-	p
1	Alpha red	Graphic red	!	1	A	Q	a	q
2	Alpha green	Graphic green	"	2	B	R	b	r
3	Alpha yellow	Graphic yellow	£	3	C	S	c	s
4	Alpha blue	Graphic blue	\$	4	O	T	d	t
5	Alpha magenta	Graphic magenta	%	5	E	U	e	u
6	Alpha cyan	Graphic cyan	&	6	F	V	f	v
7	Alpha white *	Graphic white	'	7	G	W	g	w
8	Flash	Conceal display	<	8	H	X	h	x
9	Steady *	Contiguous graphics *	>	9	I	Y	i	y
A	Nothing	Separated graphics	*	:	J	Z	j	z
B	Nothing	Nothing	+	;	K	←	k	↳
C	Normal height *	Black * background	,	<	L	↳	l	
D	Double height	New background	-	=	M	→	m	¼
E	Nothing	Hold graphics	.	>	N	↑	n	÷
F	Nothing	Release graphics *	/	?	O	#	o	■

* every line starts with these options set

Teletext graphics

	0	1	2	3	4	5	6	7
0	Nothing	Nothing						
1	Next to printer	Nothing						
2	Start printer	Nothing						
3	Stop printer	Nothing						
4	Nothing	Nothing						
5	Nothing	Disable VDU						
6	Enable VDU	Select mode						
7	Bell	Reprogram characters						
8	Back	Nothing						
9	Forward	Nothing						
A	Down	Nothing						
B	Up	Nothing						
C	Clear screen	Nothing						
D	Start of line	Nothing						
E	Paged mode	Move cursor to (0,0)						
F	Scroll mode	Move cursor						Back space and delete

	8	9	A	B	C	D	E	F
0	Nothing	Nothing						
1	Alpha red	Graphic red						
2	Alpha green	Graphic green						
3	Alpha yellow	Graphic yellow						
4	Alpha blue	Graphic blue						
5	Alpha magenta	Graphic magenta						
6	Alpha cyan	Graphic cyan						
7	Alpha white *	Graphic white						
8	Flash	Conceal display						
9	Steady *	Contiguous graphics *						
A	Nothing	Separated graphics						
B	Nothing	Nothing						
C	Normal height *	Black * background						
D	Double height	New background						
E	Nothing	Hold graphics						
F	Nothing	Release graphics *						

* every line starts with these options set

Teletext characters (used only in mode 7)
