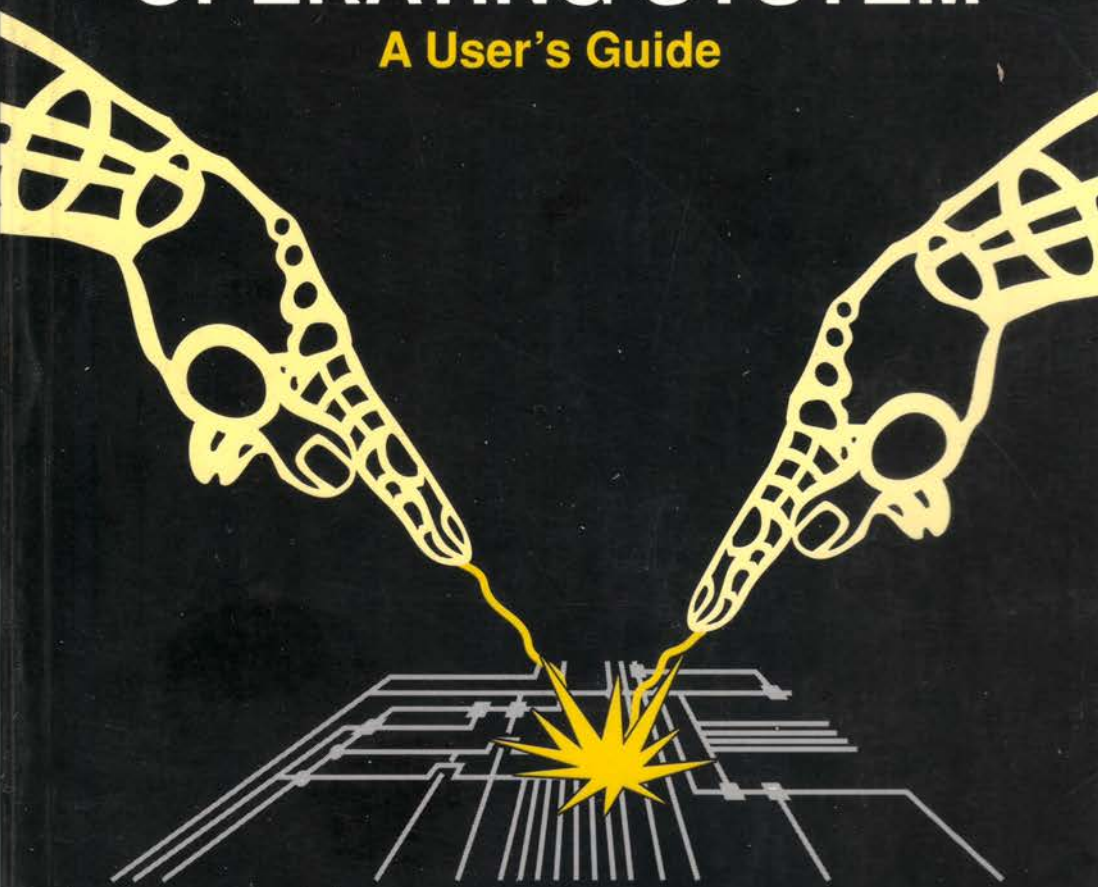
A Dabhand Guide

ALEX & NIC VAN SOMEREN

ARCHIMEDES OPERATING SYSTEM

A User's Guide



DABS PRESS

Archimedes Operating System

A Dabhand Guide

Alex and Nic van Someren

DABS PRESS

For Alice and Carol

Archimedes Operating System: A Dabhand Guide

ISBN 1-870336-48-8
First edition, second printing February 1990

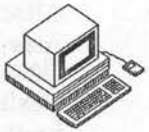
Editor: Bruce Smith
Proofreading: Syd Day
Typesetting: Bruce Smith
Cover: Clare Atherton

Published by Dabs Press, 76 Gardner Road, Prestwich, Manchester, M25 7HU. Tel. 061-773 2413.

Typeset in 10 on 11pt Palatino by Dabs Press using the Acornsoft VIEW wordprocessor, MacAuthor, Apple Macintosh SE and LaserWriter II NT.

Printed and bound in the UK by BPCC Wheaton, Exeter, Devon, EX2 8RP.

Contents



Introduction 13

Listing and Appendicies 13
What this Book is Not About 14

1 : RISC Technology 15

The History of RISC 15
RISC and the Archimedes 16
The ARM 17
The MEMC Memory Controller 18
The VIDC Video Controller 18
The IOC Input/Output Controller 18
Conclusion 19

2 : The ARM Instruction Set 20

The Programmer's Model 20
R15: The Program Counter 20
Conditional Execution 22
ARM Assembler Command Syntax 24
Shifted Operands 24
Brannch Instructions: B, BL 25
Arithmetic & Logical Instructions 26
Comparison Instructions 27
Multiplication Instructions 27
Single Register Load and Store Instructions 29
Multiple Register Load/Store Instructions 31
Software Interrupts 32

3 : The BASIC V Assembler 33

Basic Concepts 33
Using the Assembler 33
Variable Initialisation from BASIC 33

Labels in Assembler Source 34
Allocating Memory 34
Assembling Into Memory 35
Offset Assembly 35
Dealing with Forward References 35
Implementing Two-pass Assembly 36
Other Assembler Directives 37
Position Independence of Object Code 37
Executing Assembler Programs 38
Conclusion 39

4 : The Operating System 40

Communicating with the OS 40
How SWIs Work 41
SWI Names 42
SWI Error Handling 43
Error Handling – Numbering 44
Error Generation 45

5 : Command Line Interpreter 47

OS_CLI Command Syntax 47
File Redirection 48
Command Line Aliases 49
OS_CLI Commands 50
*CONFIGURE 51
*ECHO 52
*ERROR 52
*EVAL 53
*FX 53
*GO 53
*GOS 54
*HELP 54
*IF 55
*IGNORE 55
*KEY 56
*SET 57
*SETEVAL 58
*SETMACRO 58
*SHADOW 58

*SHOW 59
*STATUS 59
*TIME 60
*TV 60
*UNSET 60

6 : OS_CLI Related SWIs 61

OS_CLI 61
OS_ReadVarVal 61
OS_SetVarVal 62
Marvin 62

7 : Filing Systems 66

Introduction 66
Naming of Filing Systems, Files and Directories 67
Directories 67
Files on Different Filing Systems 68
Device Filing Systems 69
Anciliary File Information 70
Load and Execute Addresses 70
File Types and Date Stamping 71
Libraries and Search Paths 71

8 : The FileSwitch Module 74

FileSwitch Commands 74
*ACCESS 75
*APPEND 76
*BUILD 76
*CAT 77
*CDIR 77
*CLOSE 77
*COPY 78
*COUNT 79
*CREATE 80
*DELETE 80
*DIR 81
*DUMP 81
*ENUMDIR 82
*EX 82

*EXEC 83
*INFO 83
*LCAT 83
*LEX 84
*LIB 84
*LIST 84
*LOAD 85
*OPT 86
*PRINT 86
*REMOVE 87
*RENAME 87
*RUN 88
*SAVE 88
*SETTYPE 89
*SHUT 89
*SHUTDOWN 89
*SPOOL 90
*SPOOLON 90
*STAMP 90
*TYPE 91
*UP 91
*WIPE 92

9 : Filing System SWIs 93

OS_File (SWI &08) 93
OS_Find (SWI &0D) 100
File Path Considerations in OS_Find 101
Error Handling Extension 102
OS_GBPB (SWI &0C) 102
OS_BGet (SWI &0A) 105
OS_BPut (SWI &0B) 107
OS_Args (SWI &09) 107
OS_FSControl (SWI &29) 109

10 : Modules 119

Module Related Commands 119
*MODULES 120
*RMCLEAR 120
*RMKILL 120

*RMLOAD 121
*RMREINIT 121
*RMRUN 121
*RMTIDY 122
*UNPLUG 122

11 : Writing Modules 123

Workspace Memory 123
Module Errors 123
The Format of Module Code 124
Module Start-up Code 125
Module Initialisation Code 125
Module Finalisation Code 126
Service Call Handling Code 126
Service Call Reason Codes 127
Module Title String 132
Help String 132
Help and Command Decoding Table 133
Decoding Table Layout 133
SWI Chunk Base Number 136
SWI Handling Code Offset 136
SWI Decoding Table 137
SWI Decode Code 137
A Note About SWI Translations 137
OS_Module (SWI &1E) 138
Printer Buffer Module 141

12 : Writing Applications 149

Starting Applications 149
OS_GetEnv (SWI &10) 150
Alternative Ways of Starting Applications 150
Temporarily Running Another Application 151
Top Down Applications 151
The TWIN Text Editor 152
Memory Management 153
ARM Memory Structure 153
Heap Management Software and SWIs 154
OS_Heap (SWI &1D) 154
OS_ValidateAddress (SWI &3A) 156

General Guidelines on Compatibility 156
The Shell Module – Source Code 157

13 : The Window Manager 161

What's On Offer? 161
The Structure of Windows 162
Window Manager Co-ordinate System 163
Programming Using the Window Manager 163
Writing a Window Manager Application 164
The Polling Loop 166
Dealing With Reason Codes 167
Closing Down the Application Window 172
Window Manager Support for Menus 172
The Structure of Menus 173
Programming Menus 173
Menus in the WimpMaze Example 175

14 : The Font Manager 186

Dealing with the Font Manager 186
An Example 187
Getting Text on the Display 188
Plot Type Options 189
Conversions Between the Co-ordinate Systems 190
Conclusion 193

15 : Sound Introduction 195

The Three Levels of Sound 196
Level 0 – SoundDMA 196
Level 1 – SoundChannels 196
Level 2 – SoundScheduler 197

16 : Sound Star Commands 199

Level 0 Commands 199
Level 1 Commands 200
Level 2 Commands 205

17 : Sound SWI Calls 207

Level 0 SWI Commands 207
Level 1 SWI Commands 215

Level 2 SWI Commands 221

18 : The Voice Generator 226

The SVCB 226
Gate On 227
Fill 228
Gate Off 231
Update 232
Instantiate 232
Free 232
Install 232
Voice Generator Code 232

19 : Character Input/Output 238

Simple Input/Output 238
Character Input 238
Getting Hold of Single Characters 239
Whole Lines of Characters 240
Keyboard Control Functions 241
Character Output 244
Selecting Which Output Streams are to be Used 245
Selecting the VDU Stream 246
Selecting the RS423 Output Stream 246
Selecting the Printer Stream 246
Selecting the Spool File Stream 247
Character Output to the Selected Streams 248

20 : Vectors 255

The Hardware Vectors 255
The Operating System Software Vectors 258
Writing Code which Intercepts Vectors 259
SWIs Which Deal with Vectors 260

21 : Interrupts and Events 263

Good Behaviour 264
Switching Interrupts on and Off 264
The Main Interrupt Vector 265
Events 265

22 : Conversion SWIs 271

String Conversion and Decoding 271
ASCII to Binary Conversions 273
Binary to ASCII Conversions 274

23 : Miscellaneous SWIs 278

Timer Functions 278
VDU Related SWIs 281

24 : The ARM Chip Set 287

Inside MEMC 287
Virtual Memory Support 289
The MEMC Control Register 289
The Logical to Physical Translator 290
DMA Address Generators 290
Inside VIDC 291
Sound Frequency and Stereo Position 292
Inside IOC 293

25 : Floating Point Model 295

Floating Point Programmer's Model 295
The ARM Floating Point Instructions 296
Co-processor Data Transfer 297
Co-processor Register Transfer 297
Co-processor Data Operations 298
Co-processor Status Transfer Instructions 300
Conclusion 300

Appendices

A : Programs Disc 302
B : Dabhand Guides Guide 304

Index 311

Program Listings

4.1. Demonstrating OS_SWINumberToString 45
4.2. Demonstrating OS_ReadEscapeState 46
6.1. Marvin 63
9.1. Save screen using OS_File SWI 94
9.2. Use of OS_File to read catalogue information 96
9.3. Load a block of screen memory 99
9.4. Using OS_Bget to count spaces and words 106
9.5. Use of OS_FSControl to convert a file type number 113
9.6. Display directory tree 116
9.7. Manipulating file attributes 117
11.1. Printer Buffer Module 141
12.1. The Shell source 157
12.2. The Heap 159
12.3. Validate Address 160
13.1. WimpMaze 176
14.1. Font demonstration 193
17.1. Demonstrating the Sound_Configure SWI 208
17.2. Storing sounds 211
17.3. Stereo re-positioning 213
17.4. Attaching channels 220
17.5. Demonstrating the QSchedule command 222
17.6. Using QTempo and QBeat 224
18.1. Voice Generator Module Creator 233
18.2. Sound Sample 237
19.1. Simple I/O 252
19.2. WriteI example 253
19.3. PrettyPlot 253
20.1. Manipulating the hardware vectors 256
20.2. Using OS_CallAVector 261
20.3. Intercepting ReadC 262
21.1. Turning interrupts on and off 264
21.2. The bouncing mouse pointer 268
22.1. Using GS calls 272
22.2. Binary to Decimal Conversion 274
22.3. Demonstrating number conversion 276
23.1. Using timed function SWIs 279
23.2. Reading time elapsed since power-on 280

23.3. Checking screen mode 281
23.4. Cursor related SWIs 282
23.5. Reading the mouse buffer 282
23.6. Read VDU variables for given mode 284
23.7. Using OS_ReadPalette 285
23.8. Reading a pixel 286

This Book and You!

This book was written using the TWIN Text Editor on an Archimedes microcomputer. The authors' files were transfered and edited in VIEW. The completed manuscript was transferred serially to an Apple Macintosh SE where it was typeset using MacAuthor. Final camera-ready copy was produced on an Apple Laserwriter IINT from which the book was printed by A. Wheaton & Co.

Correspondence with Dabs Press, or the authors, should be sent to the address given on page 2 or via electronic mail on Telecom Gold (72: MAG11596) or Prestel (942876210). An answer to your letter or mailbox cannot be guaranteed, but we will try our best.

All correspondents will be advised of future publications, unless we receive a request otherwise. Personal details held will be provided on request, in accordance with the Data Protection Act. Catalogues detailing the full range of Dabs Press books and software are available free of charge on request.

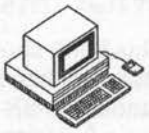
Publisher's Note

Dabs Press would like to express their thanks and gratitude to Felix Andrew for the chapters on sound and Mike Ginns for the icing!

Author's Note

The authors would like to thank David Acton for his assistance in preparing this book and David Knell for his work on the Floating Point Assembler software.

Introduction



Welcome to *'Archimedes Operating System: A Dabhand Guide'*, a book which describes the features and facilities of the Operating System for Acorn Archimedes computers. In publishing this book, our intention is to explain this rich and sophisticated piece of software which is, of necessity, rather complex. Though considerable technical documentation already exists, much of it is rather impenetrable to those who do not already understand the Operating System – this book aims to rectify the problem. In addition it supplies some vital information not published before – in particular that relating to the sound system.

Within this book you will find a sizeable part of the Operating System (OS) documented in detail – and in, what we hope, is an easily digestible form. Many examples and program listings are included and these are also available on a disc which has been produced as a companion to this guide. The programs disc also includes several extra programs and comes complete with its own User Guide. Appendix A contains full details.

Each chapter of this book describes a particular aspect of the OS. The first section is an introduction to RISC technology, and subsequent chapters describe basic Operating System functions, filing systems and it's internal structure. Two of the most useful and, we suspect, the most used components – the Window Manager and the Font Manager – are given particular attention.

Listings and Appendices

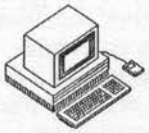
Many example listings, both short and long, are included in the relevant sections of this book. It is recommended that where an example program does not specify a display mode, an 80-column mode (such as mode 0) should be selected. Several appendices have also been included and are situated at the end of this book. These summarise information which is best left in tabular form.

What this Book is Not About

This book should not be regarded as an introduction to the Archimedes for inexperienced users: it assumes the reader has an understanding of the fundamentals of assembly language programming and is familiar with the basic functions of an Operating System. In particular, it is aimed at those familiar with the BBC Microcomputer MOS, although an experienced programmer will encounter no difficulties. Though one of the chapters of this book summarises ARM assembly language, it is by no means a tutorial introduction and you are refered to publications such as *Archimedes Assembly Language: A Dabhand Guide* by Mike Ginns and published by Dabs Press.

In attempting to explain the inner workings of the OS there is necessarily some overlap with technical details that also appear in other published works: most notably, Acorn's Programmer's Reference Manuals (PRM). It is certainly not our intention to try to replace the PRM, instead we are seeking to make this kind of information easier to understand. In practice, if you are planning on producing commercial application software or programming to an advanced level, we recommend that you refer to the PRM in conjunction with this guide: we have not attempted to cover absolutely every last detail of the OS.

1 : RISC Technology



The History of the RISC

As computer users devise increasingly sophisticated and complex applications, so the computer industry strives to provide them with more powerful machines. Countless debates about the design and architecture of computer equipment ensure that a wealth of new ideas continues to be turned into finished products. In recent years one of the best publicised of these debates has been whether microprocessors should have simpler or more complicated instruction sets to aid the writers of 'machine-code' software, in particular high-level language compilers. One camp promotes Complex Instruction Set Computers (CISCs); the other advocates Reduced Instruction Set Computers (RISCs).

The details of the RISC versus CISC debate are well outside the scope of this book, but it is worth acquiring a basic understanding of the concepts involved. Promoters of CISC architecture would like us to believe that it is most efficient to have microprocessors that execute highly sophisticated instructions – almost equal in complexity to those of the high-level languages in which most applications will be written. Clearly, if a microprocessor has instructions which multiply arrays together, for example, or instructions to extract substrings from larger sequences of characters, then the life of the compiler-writer will be apparently easier. However, such instructions are usually completed in several (and possibly many) cycles of a microprocessor's master clock and this limits the number of instructions that can be executed per second.

Advocates of the RISC philosophy, on the other hand, recommend instruction sets where every instruction performs a fairly simple task (to increment a register for example). This simplicity permits most instructions to be executed in as little as one cycle. So although we may have to use more instructions to achieve the same goal, each instruction takes the shortest possible amount of time. This approach has the disadvantage that the computer's memory needs to be capable of being addressed quite fast. Such memory is usually expensive.

While programmers may or may not like the instructions they have to use, other considerations need to be taken into account for any real application. In particular, there is the issue of interrupts. If a CISC machine is in the middle of a long and complicated instruction, it is usually impossible to halt it reliably and service an interrupt. The instruction must be completed, possibly taking a very long time, and only then may the interrupt request be dealt with. Of course, RISCs suffer rather less from this problem, which is known as 'interrupt latency', because most instructions will be completed just a few cycles after the interrupt.

Although the matter of interrupt latency is just one of many in the CISC versus RISC contest, it is one which is close to the hearts of users of Acorn computers. And close to the hearts of the architects of Acorn's products. The BBC Microcomputer (which uses the 6502 microprocessor) has the great virtue that much of its operation revolves around interrupts. This allows its users the pleasure of 'type-ahead' keyboards and background printing tasks, to name but two examples. In designing their own microprocessor, one of Acorn's goals was to ensure that interrupt latency was as low as possible, thereby allowing the strengths of the BBC Micro to be engendered in the Archimedes.

In the final analysis, each kind of architecture has certain virtues. As far as users of Acorn's RISC – Archimedes – are concerned, the important fact is that RISCs run with breathtaking speed and have an instruction set of great consistency and simplicity. These qualities make Archimedes computers a pleasure to use and to program, and with such attributes being paramount for most of us, it hardly matters whether or not CISCs are really 'better'.

RISC and the Archimedes

For a variety of reasons, Acorn was not satisfied with the RISC microprocessors being designed by other manufacturers and decided to embark upon its own RISC project. During 1983 the specification of a 32-bit microprocessor began to take shape in the minds of Acorn's senior engineers. Known as ARM, an acronym for Acorn RISC Machine, the device was conceived as the heart of the next generation of Acorn computer products. Its design owes a good deal to the 6502 microprocessor, which Acorn had used with great success for many years. In particular, both the 6502 and the ARM have very short interrupt latency (well under one microsecond on an 8 Mhz ARM). Acorn engineers were also able to go one better than other RISC manufacturers with, for example, the provision of fifteen general-purpose registers and several other sets of registers which are invisibly 'paged in' when servicing interrupts.

In Archimedes computers the ARM is supported by three other custom-built chips. These were designed by Acorn and are collectively called the 'ARM-related chip set'. Individually known as the MEMC, VIDC and IOC they are responsible for memory control, video and sound control, and input/output control respectively. This four-chip set requires very little other than RAM, ROM, a screen and a keyboard to form a complete computer. Whilst the ARM can operate without any of these chips, a desktop computer needs memory management and I/O to be useful, and these devices provide such features with an absolute minimum of extra circuitry.

The rest of this section is concerned with the ARM and the functions that each of the ARM related chips provide. If you are more concerned with programming the ARM rather than the hardware itself then you may want to skip on to the next section and perhaps read this one later.

The ARM

The ARM is a 32-bit RISC-architecture microprocessor with a full 32-bit data bus and a 26-bit address bus providing a uniform 64Mb address space. The processor is pipelined so that all parts of the system can be usefully employed in every cycle when executing register-to-register instructions. The ARM has an instruction set of 44 basic instructions. Each instruction contains a condition code that causes an instruction to be skipped if the condition is not satisfied. This allows highly efficient software to be written and has benefits for both in-line and branching sequences. The ARM contains twenty seven 32-bit registers which partially overlap, allowing the instant preservation of register contents for context switches associated with interrupt servicing.

The ARM has four modes of operation. Besides the normal 'User Mode', a 'Supervisor Mode' is provided for Operating System software. Supervisor Mode allows certain operations to be performed which are not permitted in User Mode, in particular those that directly refer to peripheral devices. The two other operating modes deal with interrupt processing. The ARM has two levels of interrupts: 'normal' and 'fast'. Interrupt servicing causes a change of processor mode as appropriate and also switches in the shadow registers for the relevant mode.

The ARM may be used in self-contained computer systems with minimal hardware support. However, three other ARM-related chips are available to workstation designers. These are discussed here.

The MEMC Memory Controller

The memory controller, or MEMC, is responsible for the interface between the ARM and the video controller (VIDC), I/O controller (IOC) and low-cost dynamic memory devices which form the main RAM.

The current version of the MEMC can support up to 4Mb of physical memory and provides all the necessary timing and refresh signals that cheap dynamic RAM chips require. The MEMC contains an address translation table which maps some of the logical address space of the ARM on to the physical memory available, simultaneously providing a three-level protection system. This assists with 'virtual memory' and allows multi-tasking operating systems to be implemented without extra hardware.

The MEMC also supports a number of Direct Memory Access (DMA) channels which are used by the VIDC to keep the display, cursor and sound channels running with minimal processor intervention.

The VIDC Video Controller

The video controller, VIDC, is a combined video and audio processor. It enables the generation of video images at a number of different pixel resolutions and bits-per-pixel. The VIDC also contains a colour mapping palette to allow the displayed colours to be selected from a range of 4096 possible hues.

Stereo sound is also generated by the VIDC and filtered by off-chip electronics into two channels of high-quality audio.

The current version of the VIDC is capable of generating colour displays of up to 640 by 512 pixels of four bits-per-pixel. A small amount of extra circuitry (fitted to Archimedes 400-series computers) allows a 1280 by 960 monochrome display to be produced, rivalling the image quality of engineering workstations in the £20,000 price region.

The IOC Input/Output Controller

The input/output controller, or IOC, provides peripheral and interrupt control signals with programmable timing parameters to suit most applications.

The IOC contains four independent 16-bit programmable counters – configured as two timers and two baud-rate generators (for RS232 and keyboard communications). A bi-directional serial keyboard interface is also included.

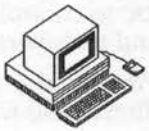
Four programmable types of peripheral access timing are generated by the IOC to control 'podules'. The IOC also deals with interrupt requests, masks, and peripheral and podule status.

Finally, the IOC provides a number of direct control lines (some bi-directional) which may be used to drive peripherals and I/O circuitry.

Conclusion

The ARM and its related chip-set form a very powerful 32-bit computer core which takes up less than ten square inches of circuit board space. When coupled with RAM, ROM, disc drives and display electronics, a complete workstation can be constructed with a component count which would, just a few years ago, have been inconceivably low. It is a tribute to Acorn's design staff that this impressive array of devices is available well in advance of most competitors' RISC micros, let alone support circuits.

2 : The ARM Instruction Set



This chapter provides an introduction to the ARM instruction set and to the assembly language in which it is programmed. If you already have experience of ARM assembly language programming or you have read *Archimedes Assembly Language: A Dabhand Guide* then you will probably want to skip on to the next two chapters. We include a brief synopsis here for those who have little or no prior experience of ARM assembly language.

An appendix at the back of this book discusses the extensions to the ARM instruction set which allow floating-point operations to be coded in assembly language. Because the assembler built into the BASIC V interpreter does not cater for these instructions we have included a program on the disc which accompanies this book to allow such instructions to be assembled.

The Programmer's Model

The ARM provides sixteen 32-bit registers which are usually known as R0 to R15. In fact, a number of 'shadow' registers are provided in the processor and these are switched in automatically when the processor mode changes. This means that interrupt service routines don't need to save registers explicitly and can thus be executed more quickly. So while only 16 registers are accessible to the programmer at any one time, a total of 27 are actually provided.

The 'programmer's model' of the registers is shown in the figure 2.1 on the next page.

R15: The Program Counter

Unlike R0 to R14, R15 is not a general-purpose register; instead, it contains the Program Counter (PC) and the Processor Status Register (PSR). The program counter occupies 24 bits of the register, with the remaining eight bits used as the PSR. Six of the PSR bits contain status flags and the remaining two indicate which of the four processor modes prevails. The exact arrangement of bits is illustrated in figure 2.2 overleaf.

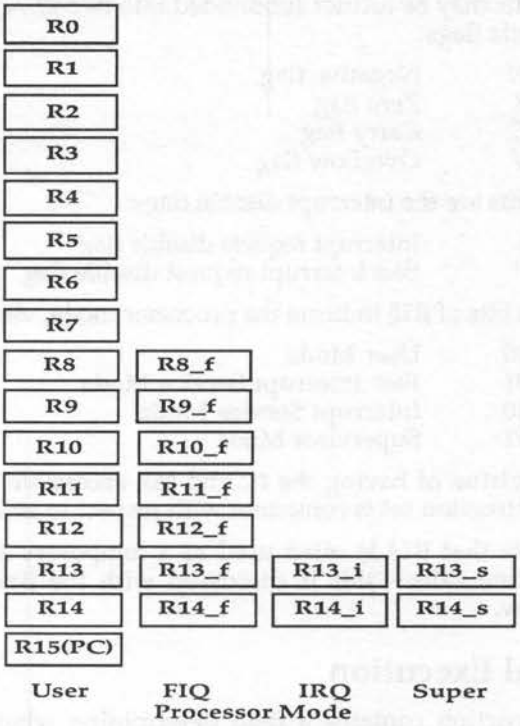


Figure 2.1. ARM programmer's model – register map.

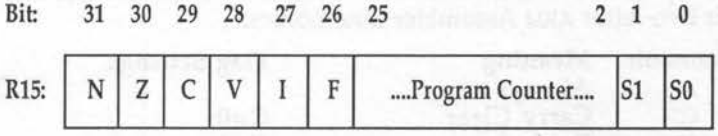


Figure 2.2. R15 – arrangement of bits.

The processor views memory as a linear array of bytes but accesses memory on four-byte units known as 'words'. ARM instructions, pointed to by the PC, are 32 bits long and must be 'aligned' within one memory word. Thus although the processor address space is 2^{26} bytes (or 64Mbs),in fact

only 24 bits are required because word-alignment renders the bottom two bits unnecessary.

Six of the flag bits may be further subdivided into two groups. The top four bits are arithmetic flags:

N	Negative flag
Z	Zero flag
C	Carry flag
V	Overflow flag

The other two bits are the interrupt disable flags:

I	Interrupt request disable flag
F	Fast interrupt request disable flag

The bottom two bits of R15 indicate the processor mode, viz:

00	User Mode
01	Fast Interrupt Service Mode
10	Interrupt Service Mode
11	Supervisor Mode

Of course, the virtue of having the PC and PSR accessible through R15 is that the ARM instruction set is consistent with respect to all registers.

You should note that R14 is often used as a temporary store for the PC during subroutine calls – this is discussed with the Branch with Link instruction below.

Conditional Execution

Each ARM instruction contains a field determining whether or not the instruction is executed, depending on the state of the arithmetic flags N, Z, C, and V. In addition, there are the two conditions 'Always' and 'Never'. The sixteen possible condition combinations are shown in table 2.1, along with their two-letter ARM Assembler mnemonics:

Mnemonic	Meaning	Flag Settings
AL	ALways	
CC	Carry Clear	C=0
CS	Carry Set	C=1
EQ	EQual	Z=1
GE	Greater than or Equal	N=1 and V=1 or N=0 and V=0
GT	Greater Than	N=1 and V=1 and Z=0 or N=0 and V=0 and Z=0

Mnemonic	Meaning	Flag Settings
HI	HIgher (unsigned)	C=1 and Z=0
LE	Less than or Equal	N=1 and V=0 or N=0 and V=1 or Z=1
LS	Lower or Same (unsigned)	C=0 or Z=1
LT	Less Than	N=1 and V=0 or N=0 and V=1
MI	MInus (Negative)	N=1
NE	Not Equal	Z=0
NV	NeVer	
PL	PLus (Positive)	N=0
VC	oVerflow Clear	V=0
VS	oVerflow Set	V=1

Table 2.1. Conditional mnemonics.

Two synonyms are also in common use and are supported by the assembler:

LO LOwer (unsigned) is equivalent to Carry Clear (CC)
HS Higher or Same (unsigned) is equivalent to Carry Set (CS)

ARM Assembler Command Syntax

In the discussion of the ARM instructions in the rest of this section, the following syntax is used to indicate instruction components:

{}	indicates optional components
(a b)	indicates that either a or b but not both may be used
#	introduces an immediate value (see below also)
#<exp>	indicates an expression which evaluates to a constant
Rd	indicates the destination register for the result
Fd	floating point destination register
Rx, Ry, Rz	indicate source registers
Fx, Fy	floating point source registers
shift	indicates that one of the following options should be used:

ASL (Rs #<exp>)	Arithmetic shift left by contents of Rs or expression
LSL (Rs #<exp>)	Logical shift left etc.
ASR (Rs #<exp>)	Arithmetic shift right etc.
LSR (Rs #<exp>)	Logical shift right etc.
ROR (Rs #<exp>)	Rotate right etc.
RRX	Rotate right by one bit with extend

Shifted Operands

The ARM has a 32-bit 'barrel shifter' which is capable of shifting an operand an arbitrary number of bit positions before using it in an instruction. This leads to two possible shift instruction types for the second operand of data processing instructions:

1. The shift is specified by using an extra register, Rs, given after the shift operation mnemonic. This specifies that the instruction's operand is to be shifted by the number of places contained in the given register. For example, if the shift register, Rs, contained 27, then the operand would be shifted 27 places before being used by the instruction.

2. The shift is specified by using an immediate constant. Here the absolute number of places by which to shift the operand, is given as a immediate constant number preceded by a '#' as normal.

Note: It is pointless to try to attempt to shift an instructions operand if the operand is given itself as an immediate constant. Shifts are only valid, therefore, when the second operand to an instruction is given in a register. No matter how a data shift is specified, the maximum number of places to shift by is always 31.

Branch Instructions: B, BL

Syntax:

```
B{L}{cond} <expression>
```

Like all other ARM instructions, the following instructions are only executed if the condition field is true. The ARM assembler assumes a condition of ALways (ensuring execution) unless you specify otherwise.

There are two different kinds of branch instruction: B (for Branch) and BL (for Branch with Link). The mnemonics are chosen to produce conditional instructions which are similar to their equivalent 6502 instructions, eg, BEQ for Branch if EQual.

Pure branches simply perform a relative branch to the specified offset if the condition field of the instruction is true. Thus, a minimal (and quite useless) loop is assembled as follows:

```
.label  BAL label
```

Unlike the 6502, the ARM branch instruction takes an offset which is almost large enough to encompass the entire address space of the processor. You will recall that this is 64Mbs or 2^{26} bytes, but because instructions must be word-aligned, only 24 bits are required in the instruction. This means that branch instructions can reference *any* word in the entire 64Mb address space and 'out of range' errors cannot occur.

Branch with Link (BL)

The Branch with Link instruction differs from the Branch instruction in that it provides a return address for the branch, ie, a subroutine facility. Before the branch is made (if the condition field allows) the PC is adjusted and saved into R14 to provide a 'link' to the address of the instruction after the Branch with Link.

To return from a Branch with Link several possible mechanisms are available according to two criteria: whether the PSR should be restored or not and whether R14 is still valid or needs to be retrieved from a stack. The former affects whether subroutines can alter the flags and the latter is

necessary to support more than one level of subroutine calling. These mechanisms are outlined below:

Return through R14 and restore PSR	MOVS PC,R14
Return through R14 (PSR unaltered)	MOV PC,R14
Return popping address from stack and restore PSR	LDMFD Ry!,{PC}^
Return popping address from stack (PSR unaltered)	LDMFD Ry!,{PC}

where Register 'Ry' is acting as the stack pointer. These instructions will make a little more sense when you have read the whole chapter!

Arithmetic and Logical Instructions

Syntax:

```
<mnemonic>{cond}{S} Rd,Rx,(Ry{,shift} #<exp>)
```

or where instruction requires only one source parameter:

```
<mnemonic>{cond}{S} Rd,(Ry{,shift} #<exp>)
```

These instructions perform fundamental data processing – movements and operations between registers. They are listed below in table 2.2:

Mnem.	Function	Result using Rd, Rx, Ry
ADC	Add with carry	Rd=Rx+Ry+C
ADD	Add without carry	Rd=Rx+Ry
SBC	Subtract with carry	Rd=Rx-Ry-(1-C)
SUB	Subtract without carry	Rd=Rx-Ry
RSC	Reverse subtract with carry	Rd=Ry-Rx-(1-C)
RSB	Reverse subtract without carry	Rd=Ry-Rx
AND	Bitwise logical AND	Rd=Rx AND Ry
BIC	Bitwise logical AND NOT	Rd=Rx AND (NOT Ry)
ORR	Bitwise logical OR	Rd=Rx OR Ry
EOR	Bitwise logical EOR	Rd=Rx EOR Ry
MOV	Move	Rd=Ry (Rx not used)
MVN	Move NOT	Rd=(NOT Ry) (Rx not used)

Table 2.2. The arithmetic and logical instructions.

All of these instructions place the result in the destination register 'Rd' without affecting memory in any way.

In addition, each of these instructions may use a shifted operand as described earlier.

The 'S' option in the instruction controls whether the PSR flags are affected by the result of the operation. When 'S' is present the arithmetic operations affect the N, Z, C and V flags, and bitwise logical operations affect the N, Z and C flags (C being affected if shifted operands are used).

If R15 is specified as the destination register then 'S' is used to decide whether all 32 bits of R15 are updated (if S is present) or just the 24 PC bits (if S is not present). Note that in User Mode the mode bits and the I and F flags cannot be altered. Where R15 is used as a source operand, if R15 is used as the first operand (Rx) only the 24 bits of the PC are available, but when employed as a second operand (Ry) all 32 bits are available.

Comparison Instructions

Syntax:

```
<mnemonic>{cond}{P} Rx,(Ry{,shift} #<exp>)
```

The comparison instructions are much the same as the arithmetic and logical instructions except that they always set the flags (ie, 'S' is always implied) and they do not return a result.

The comparison instructions are as follows:

Mnem.	Function	Arithmetic/logic used to set flags
CMN	Compare (negative)	Rx+Ry
CMP	Compare	Rx-Ry
TEQ	Test equal bitwise	Rx EOR Ry
TST	Test bitwise	Rx AND Ry

Besides their fundamental function of setting the flag bits according to the result, the comparison instructions may also be used to force the PSR bits to a specific state by means of the 'P' option. When present, the 'P' option causes the top six bits and the bottom two bits of the result to replace the PSR flags and mode bits in the corresponding positions.

Multiplication Instructions

Syntax:

```
<mnemonic>{cond}{S} Rd,Rx,Ry{,Rz}
```

The ARM multiplication instructions (detailed below) perform integer multiplication on two full 32-bit operands, giving the least significant 32

bits of the result. The Multiply and Add instruction also adds the third operand into the result last of all.

Mnemonic	Function	Arithmetic used
MUL	Multiply	Rd=Rx*Ry
MLA	Multiply and add	Rd=(Rx*Ry)+Rz

The PC may not be used as the destination register (a quite meaningless operation after all). Nor may the destination register Rd be the same as Rx. The 'S' option controls whether the N and Z flags are set by the result. The V and C flags are unchanged and undefined, respectively.

A full 64-bit result may be generated with the following sequence:

```
.fullmultiply
;Replace x,y,p,q and t by register numbers
;pq = x * y
;Register t is used for temporary space

;Break x and y into 16bit numbers
MOV t,x,LSR #16
MOV p,y,LSR #16
BIC x,x,t,LSL #16
BIC y,y,p,LSL #16

;Multiply the chunks together in pairs
MUL q,x,y
MUL y,t,y
MUL x,p,x
MUL p,t,p

;Add the products up,
;with careful consideration for carry's
ADDS x,y,x
ADDCS p,p,#&10000
ADDS q,q,x,LSL #16
ADC p,p,x,LSL #16
```

The ARM does not provide a division instruction, but a simple section of code to achieve this is listed below:

```
.divide
;Divide R0 by R1
CMP R0,R1                ;Test if result is zero
MOVMI R0,#0              ;If it is, give result *
MOVMI PC,R14             ; and return
CMP R1,#0                ;Test for division by zero
ADREQ R0,divbyzero       ; and flag an error
SWIEQ "OS_GenerateError" ; when necessary
STMFD R13!,{R2,R3}
MOV R2,#1
```

```
MOV R3,#0
CMP R1,#0
.raiseloop
BMI raisedone
CMP R1,R0
BHI nearlydone
MOVS R1,R1,LSL #1
MOV R2,R2,LSL #1
B raiseloop
.nearlydone
MOV R1,R1,LSR #1
MOV R2,R2,LSR #1
.raisedone
CMP R0,R1
SUBCS R0,R0,R1
ADDCS R3,R3,R2          ;Accumulate result
MOV R1,R1,LSR #1
MOVS R2,R2,LSR #1
BCC raisedone
MOV R0,R3               ;Move result into R0 *
LDMFD R13!,{R2,R3}
MOV PC,R14              ; and return

.divbyzero ;The error block
EQUD 18
EQUS "Divide by Zero"
EQUB 0
ALIGN

; * Remove the lines marked with asterisks to
; return R0 MOD R1 instead of R0 DIV R1
```

Single Register Load and Store Instructions

Syntax:

```
<mnemonic>{cond}{B}{T} Rd, <address>
```

Although the ARM has a large number of internal registers for efficiency, it is of course necessary to have some way of loading and storing the contents of these registers from and to main memory. It is a tenet of RISC philosophy to minimise the complexity of these instructions, and the ARM is no exception. The ARM load and store instructions are shown below:

Mnemonic	Function
LDR	Load register
STR	Store register

Both of these instructions deal with a register (to be loaded or stored) and an address calculated with one of seven addressing modes.

The whole 32 bits of the register or memory location concerned are affected unless a 'B' is added to the mnemonic in which case only a single byte is transferred.

The simplest addressing mode takes a register number, the specified register containing the address to load from or store to. This is known to users of most other microprocessors as 'register indirect' addressing.

Other options include adding an immediate value or the contents of another register to the address contained in the first register. This is known as pre-indexed addressing since the calculation of the final address is performed before the load or store takes place. The register may optionally be updated with the result of the address calculation by appending an exclamation mark '!' after the addressing mode.

The syntax of the pre-indexed address modes is as follows:

Address calculation	**Syntax**
Contents of Rx	[Rx]
(Contents of Rx)+m	[Rx,#m]{!}
(Contents of Rx)+(contents of Ry)	[Rx,Ry]{!}
(Contents of Rx)+ (contents of Ry, shifted by s bits)	[Rx,Ry,shift #s]{!}
(Contents of Rx)+ (contents of Ry, shifted by the number of places held in Rs)	[Rx,Ry,shift Rs]{!}

Alternatively, the address calculation may take place after the load or store, such addressing being known as post-indexed addressing. Post-indexed addressing automatically writes the result back into the specified register, so an exclamation mark is never needed.

The syntax of the post-indexed addressing modes is shown below:

Calculation performed and written back	**Syntax**
(Contents of Rx) – then increment Rx by 'm'	[Rx],#m
(Contents of Rx) – then increment Rx by contents of Ry	[Rx],Ry
(Contents of Rx) – then increment Rx by contents of Ry, shifted by s bits)	[Rx],Ry,shift #s

Remember that, in all post-indexed addressing modes, the data is obtained from the address held in Rx alone. It is only after this has happened that the contents of Rx are changed by adding the suitably specified offset. In

pre-indexed addressing, this modification of the address in Rx takes place first. The data is then loaded from the newly modified address. Finally, if write back is selected, the new modified address, just used, is stored back in Rx, replacing its original contents.

If you give the assembler a simple expression as the address it will generate a pre-indexed instruction using R15 (PC) as the base register, thus providing 'position-independent' assembly. An error will be generated if the address is outside of the range of the instruction (+ or – 4095 bytes).

Note also that it is not possible to use a register to specify a shift amount with the LDR and STR instructions.

If 'T' is added to the mnemonic the MEMC memory controller is forced to cause an address translation in Supervisor Mode (which would not usually happen). In User mode address translation takes place all of the time so this is unnecessary.

Multiple Register Load/Store Instructions

Syntax:

```
<mnemonic>{cond}(I D)(A B) Rx{!},<Rlist>{^}
```

When writing subroutines and procedures it is frequently necessary to preserve the contents of several registers; usually this is achieved by pushing them on to a stack so that they may be popped off later. To save (programming) time the ARM provides instructions for loading and storing any or all of the internal registers. In fact, one instruction suffices for each operation because individual bits in the instruction indicate which registers are to be dealt with.

The multiple register instructions are summarised below:

Mnemonic	Function
LDM	Load multiple registers
STM	Store multiple registers

The contents of register Rx are used as the base address for the load or store operation. The list of registers 'Rlist' will be loaded from or stored to memory, starting with the lowest numbered register.

Two mandatory components of the instruction affect its operation: the (ID) field controls whether addresses loaded from or saved to are Increased or Decreased from the base address in Rx. It is therefore possible to create stacks which extend either upwards or downwards in memory.

The second control field (A B) indicates whether the address used is modified after or before the load or store operation. If 'A' is used, the first register is dealt with and then the address is updated before the next register. Alternatively, using 'B' causes the address to be updated before each register is dealt with. To simplify the programmer's life, synonyms are available for these two control fields: an 'F' (for 'Full stack') may be used in place of 'B' and an 'E' (for 'Empty stack') may may be used instead of 'A'. Similarly, the second letter may be replaced with 'D' for 'Descending' or 'A' for 'Ascending'. All Acorn software uses Full Down (FD) stacks and it is recommended that you follow suit.

Software Interrupts

Syntax:

```
SWI <expression>
```

The remaining ARM instruction is the Software Interrupt (SWI). The purpose of the SWI instruction is to allow the Operating System to make its facilities available to the user without allowing the user any direct control in Supervisor Mode. On encountering this instruction, ARM changes to Supervisor Mode (thus preserving the user's R13 and R14 registers in their own register bank) and jumps through the SWI vector to allow the instruction to be processed. The assembler evaluates the expression that follows the SWI mnemonic into a 24-bit field which is used to determine the action required.

Both the Arthur and RISC OS Operating Systems use SWIs as the fundamental control mechanism for user software. The 24-bit SWI number is decoded and used to select the Operating System routine to be performed before returning to the next instruction of the user's program. We shall see a great deal more of SWIs throughout this book.

3 : The BASIC V Assembler

Basic Concepts

Having examined the instructions available to ARM programmers we need to devote some attention to the ARM assembler which allows us to create machine code programs from the keyboard.

Of course, it would be possible to hand assemble the instructions for our program by looking up their opcodes as numerical values and composing complete instructions. However, quite aside from the effort involved in this process, the complexity of the instructions at the individual bit level is such that only a real masochist would attempt this more than a couple of times.

Instead, Acorn has followed the tradition started by the BBC Microcomputer and included an assembler in the BASIC interpreter. This is to Acorn's benefit because the assembler can take advantage of many of the housekeeping facilities that BASIC needs for itself; it is also to the programmer's benefit for much the same reasons!

Most readers will be familiar with the 6502 assembler provided in BBC BASIC for the BBC Microcomputer. Whether you are or not, it is worth reading the synopsis that follows because some significant extensions are present in the ARM version.

Using the Assembler

In common with the 6502 assembler the ARM assembler is entered when ARM BASIC encounters an opening square bracket ' ['. Similarly, a closing square bracket '] ' exits the assembler and returns control to the BASIC interpreter.

Variable Initialisation from BASIC

BASIC and the assembler are inexorably intertwined in that they share the same variables and workspace. Thus it is possible to initialise variables to useful values from BASIC and then employ them in the assembler. For example, to create a constant we might say:

```
Screen_mode=128
```

and thereafter employ 'Screen_mode' throughout our assembler program. This has the virtue that we need only alter this one variable and re-assemble our software to update all its occurences.

Labels in Assembler Source

The assembler takes advantage of BASIC in a similar way to allow you to use alphanumeric labels in the assembler program. Each location in the program which needs to have a label is marked by preceding it with a full stop. For example:

```
.my_label MOV R0,R1
```

labels the address at which the 'MOV R0,R1' instruction is stored as 'my_label' to allow branches and references to it elsewhere.

The assembler does this by creating a BASIC variable of the same name, ie, my_label, and setting it to the value of P% (see later) at the time the label was encountered. When a reference to the label is necessary elsewhere, it may simply be referred to by name, eg:

```
BLEQ my_label
```

Again, this has the virtue that only the label needs to be moved before re-assembly in order to update all references to it.

Allocating Memory

The ARM assembler deposits the object code it generates into memory at a defined place (see later). In order to prevent BASIC and the assembler from attempting to make conflicting use of this memory we need to advise BASIC that it should not be corrupted. This is achieved using a special case of the BASIC 'DIM' operator. To reserve 1000 bytes for assembler object code we might use the following:

```
DIM object_code% 1000
```

which reserves at least 1000 bytes and leaves the BASIC variable 'object_code%' pointing to the first byte. Note that memory reserved in this way is guaranteed to be word aligned, ie, its start address will always be on a word boundary (address divisible by four).

You may find this mechanism useful for reserving space for tables or to allow you to refer to large amounts of data from the assembler.

Assembling Into Memory

Once you have reserved an area of memory into which the assembler may put the object code it produces, you must advise the assembler of the location of the memory. The assembler shares the BASIC variables P% and O% for this purpose.

P% is used to point to the start of the memory area, eg:

```
P%=object_code%
```

When assembly begins, the first instruction assembled will be placed at P% and the value of P% will then be inceased by four automatically after assembling each instruction so that it points to where the next instruction should be placed. Thus, when the assembly is complete, P% will be left pointing to the byte after the last instruction assembled.

Offset Assembly

The use of P% as a pointer has the drawback that you must assemble your code at the position where it will ultimately be used. This is not always convenient. Programs which need to be executed at a different address to that at which they are assembled may contain references to memory locations which are fixed at assembly time. To allow for this, the variable O% may be used to set the address at which the object code is to be stored. The assembler responds by placing the object code in memory starting at O% but uses P% to resolve any absolute memory references in the object code. O% and P% are incremented together in this case.

Of course, you need to advise the assembler that you wish the object code generated to be executed in a different place – this is achieved by setting bit two of the assembler directive OPT and is discussed in more detail below.

Dealing With Forward References

As we saw above, the assembler allows labels to be placed in the source program to make the program easier to understand. However, it is possible for a situation to arise in which the assembler cannot immediately resolve all references to labels. Consider the following program fragment:

```
B forward_label
...
.forward_label
```

which on its own would cause an assembler error ('Unknown or missing variable'). As you can see, there is a reference to a label which has not yet been encountered by the assembler and whose address is therefore unknown. To overcome this we can perform the assembly twice, discarding the results (and errors) from the first attempt. This is known as 'two-pass' assembly and should be used in preference to other approaches because it covers all eventualities.

The assembler directive OPT is used to control the assembly, particularly with regard to forward reference errors and where the object code is to be stored. The bottom three bits of the value following an OPT control these assembler features and are used as follows:

Bit	Effect if set
0	Assembler displays assembly listing
1	Assembler reports assembly errors
2	Assembler places code at O%, not P%

The listing flag (bit 0) simply controls whether the assembler displays the program as it assembles; it has no effect on its execution.

The assembly error flag (bit 1) allows you to suppress errors during the first pass of two-pass assembly. It is usually cleared for the first pass (to suppress errors) and set for the second pass (to show up errors not concerned with forward references).

The code destination flag (bit 2) has been discussed above and controls whether the object code is placed at P% (if clear) or O% (if set).

Implementing Two-pass Assembly

The simplest way to implement two-pass assembly is to enclose the entire assembler part (between square brackets) within a FOR...NEXT loop which changes OPT as necessary. Typically it would look something like this:

```
FOR pass%=0 TO 3 STEP 3
P%=object_code%
[OPT pass%
...assembler program here
]
NEXT pass
```

Notice that this is the simple case where the object code is placed at P%. The modifications required to support assembly to O% are left as an exercise to the reader.

Other Assembler Directives

The ARM BASIC V assembler supports some other useful directives for saving time when writing and assembling programs. These are summarised below:

Directive	Effect
EQUB value	Allocates one byte and pokes (value MOD &FF)
EQUW value	Allocates two bytes and pokes (value MOD &FFFF)
EQUD value	Allocates four bytes and pokes (value)
EQUS string	Sets aside as many bytes as neccessary and puts the ASCII string in them.
ALIGN	Increments P% and O% to align them to the next word boundary
ADR reg,address	Assembles an instruction to load 'address' into register 'reg'

The four 'equates' set aside the specified number of bytes from P% and initialise them as described. EQUB, EQUW and EQUD take an expression which evaluates to an integer, whilst EQUS should be followed by a string expression. Because strings are rarely a multiple of four bytes in length it is neccessary to ensure that P% and O% are aligned on a word boundary before continuing the assembly (remember that instructions must be word-aligned). A typical sequence for initialising a zero-terminated string is thus:

```
...
EQUS "This is a string"
EQUB 0
ALIGN
...
```

Position Independence of Object Code

In order that ARM assembler programs should be as versatile as possible it is good practice to make them 'position independent'. This is to say that the program contains no 'absolute' or fixed references to itself, for example, tables or other data. In fact, a requirement is that extensions to the Operating System (known as 'modules') be position independent because it frequently relocates them in memory according to circumstances.

The simplest way to achieve position independence is to ensure that all references are given relative to the PC. It is with this in mind that the ARM branch instructions are all relative rather than absolute.

The assembler directive ADR assists by automatically generating an ADD or SUB instruction which will ensure that the specified register contains a position independent pointer to the specified address (by using the PC in its calculation). Consistent use of this directive allows you to ensure that your software is position independent.

An acid test for determining whether software is truly position independent is to assemble it at two different locations and compare the listings produced using a text editor or file comparison program. By scrutinising the differences between the two listings it is possible to identify instructions which need re-writing.

Executing Assembler Programs

Once a program has assembled correctly it is usually a good idea to save the object code! The length of the object code can be determined by subtracting the start address from P% (or O%), eg:

```
PRINT"Object code starts at "~objcode_code%" and is "~P%-
objcode_code%" bytes long."
```

This information is enough to allow you to save it using the operating system command '*SAVE'. The simplest syntax of this command is:

```
*SAVE<filename><startaddr><endaddr>{<executeaddr>{<reloadaddr>}}
```

or alternatively:

```
*SAVE<filename><startaddr>+<length>{<executeaddr>{<reloadaddr>}}
```

where all of the address-related information is assumed to be entered in hexadecimal and the execute and reload addresses are optional.

Once the object code has been saved it may be executed in several ways. Firstly, it may be executed in memory without reloading by means of one of two BASIC commands:

```
CALL startlabel
```

or alternatively:

```
variable=USR(startlabel)
```

with 'startlabel' being either the name of the entry point or its address in hexadecimal. Either of these commands will start executing the object code at the specified address, hopefully producing the expected results!

Alternatively, the saved object code may be executed by re-loading it and running it. The operating system deals with loading automatically if you

type any of the following three synonymous commands to run the object code:

```
*<filename>
```

or:

```
*RUN <filename>
```

or:

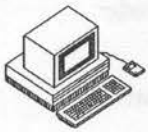
```
*/<filename>
```

For more information on *SAVE and its address parameters consult the *Archimedes User Guide* or *Programmer's Reference Manual.*

Conclusion

The ARM BASIC V assembler allows programs to be written quickly and easily by providing the programmer with a familiar environment in which to work (BASIC) and extending it very slightly to cater for the quirks of assembler programming. This successful formula was adopted with the BBC Microcomputer and has been updated to deal with the complexities of the ARM.

4 : The Operating System



The 'Operating System' is the name given to the fundamental software in a computer which provides the environment in which the user works. Without an Operating System it would not be possible to type commands at the keyboard, see those commands on the display or even execute programs. The Operating System relieves the user of the complexities of controlling the various peripheral devices connected to the computer and allows the computer to be operated with meaningful commands rather than by machine-code programming. At the same time, the Operating System software is broken up into many distinct routines which may be called individually, so as to allow programmers to take advantage of useful facilities that have already been written.

The most crucial role of the Operating System is in controlling the input/output devices connected to the computer: in particular, the keyboard, screen display and storage devices such as disc drives. This is achieved with the help of hardware 'interrupts' – signals from peripherals which indicate that they need attention from the computer. If you have programmed in assembly language on a BBC Microcomputer then you are likely to be aware of the importance of interrupts. They are equally important in the Archimedes and, as we saw in the introduction, the ARM is very competent at servicing interrupts quickly.

The Acorn Archimedes computers are sophisticated and powerful machines which are designed to be expanded by the addition of new software and new hardware. Clearly, the Acorn staff responsible for the design of the computer cannot think of every possible extension which their myriad users might devise, so instead they concentrate on making the Operating System as general as possible while allowing it to be expanded with ease.

Communicating with the OS

There are two mechanisms by which users can communicate their wishes to the Operating System. The more familiar of the two is the keyboard 'command line' which interprets commands beginning with *. The OS displays an asterisk whenever it is ready for a command to be typed by the user for execution, and for this reason its prompt symbol is itself an

asterisk (useful because it saves the user from having to type one). When this prompt is visible, any of the Operating System commands (or commands supported by other resident software) may be entered for execution. The Operating System will respond either by taking the specified action or reporting an error if the command was not understood. Because the command line is so crucial to the operation of the Operating System we shall examine it in greater detail later.

The second way in which communication with the Operating System can take place is through the use of 'software interrupts' or 'SWIs'. As we saw in the previous chapter, the ARM has an instruction known as SWI which effectively acts as a subroutine call into the Operating System. This instruction always takes a 24-bit parameter to indicate which operation is desired. It may also require other parameters to be placed in the ARM registers according to the operation.

The reason that these two different techniques exist is simply one of efficiency. For human beings, writing a short piece of assembler every time we want the computer to do something is seriously inefficient. On the other hand, the converse is true for programs (where composing and issuing long text strings is equally inefficient – particularly in terms of memory usage). Thus, the command line exists to make it easier for the user to enter quick commands, while the SWI instruction is used to achieve all these effects and others from within programs.

It is worth noting that in fact the 'command line interpreter' (CLI) is actually an Operating System routine which may be called by means of a SWI, so in reality there is only one way of communicating with the Operating System, viz using SWIs. However, it will be convenient to think of the two as being distinct.

How SWIs work

The 24-bit SWI identification field is large enough to allow just over sixteen million different SWIs to be specified. In practice, this field is divided up into several groups to allow different kinds of SWI to be numbered in related ways. The meaning of the individual SWI number bits is as follows:

Bits 23-20 These top four bits are used to identify the Operating System of which the SWI is a part. All four bits must be set to be zero to indicate that the SWI is relevant to Arthur and RISC OS; other Operating Systems will use different values.

Bits 19-18 These two bits are used to indicate which piece of software is responsible for executing the SWI. The table below summarises the possibilities:

Bit 19	Bit 18	Meaning
0	0	Operating System
0	1	Operating System extensions (Acorn)
1	0	Third party applications
1	1	User programs

Bit 17 This bit, known as the 'X' bit, is used to specify how errors that occur during the execution of the SWI should be dealt with. It is discussed in more detail in the section on SWI error handling.

Bits 16-6 These eleven bits identify the group or 'chunk' of 64 SWIs for each specific application. Chunk numbers are allocated by Acorn as new software requires. A few SWI chunks have already been allocated to various parts of the Operating System and filing systems. If you produce commercial software which requires several SWIs you should apply to Acorn for a chunk number.

Bits 5-0 The bottom six bits indentify the particular SWI within a given chunk. This allows up to 64 SWIs (which is more than enough) for each application.

SWI Names

Obviously, it would be inconvenient to have to remember or look up a 24-bit number every time you wanted to call a SWI from a program. To save time, the Operating System provides a mechanism for giving SWIs textual names and then converting between SWI names and SWI numbers. Two SWIs, whose textual names are:

OS_SWINumberToString
OS_SWINumberFromString

allow the Operating System to perform these conversions. Since the Operating System cannot pluck textual names from thin air, the author of a particular SWI is obliged to follow a documented standard to allow these conversions to take place. This is discussed in the section on 'modules'.

Listing 4.1 uses OS_SWINumberToString to print out the names of the first 256 SWIs. OS_SWINumberToString returns each SWI name at the location called buffer% and the name is printed using OS_Write0.

Both the ARM BASIC command 'SYS' and the ARM BASIC assembler perform this conversion automatically whenever they encounter a SWI name enclosed in inverted commas.

In order to make these conversions somewhat simpler to implement, the SWI chunk 'name' followed by an underline '_' is used to prefix the SWI, so the conversion SWIs are named 'OS_...' because they are provided by the Operating System. SWIs provided by other software modules do the same, so the ADFS SWIs are all prefixed by 'ADFS_...' and so forth.

Note that SWI names must be spelt exactly as seen, so capital and lower-case letters are crucial to correct SWI calling. If you mis-spell the SWI name an error will be generated. We can only hope that in future versions of the Operating System case independence and abbreviation of SWI names will be permitted.

SWI Error Handling

Almost every SWI needs to deal with circumstances where an error may arise, such as where it is passed insufficient or erroneous parameters or because the action it tried to take failed for some low-level reason. Two mechanisms for dealing with SWI errors are defined by the Operating System so that programmers may take appropriate action.

In its simplest form, SWI error handling is achieved through the use of the ARM's Overflow flag 'V'. All SWIs indicate their successful completion by clearing this flag, a result which is easily dealt with by means of a branch instruction after the SWI instruction. If an error arises, the SWI sets the 'V' flag and returns with ARM register R0 containing a pointer to a block of information describing the error. SWIs which behave in this way are known as 'error-returning'.

The format of the error block pointed to by R0 for error-returning SWIs is:

Bytes 0-3	Error number
Bytes 4-n	Error message (ASCII text)
Byte n+1	Zero (to terminate the error message string)

Error blocks must be word-aligned and may not exceed 256 bytes in length.

The more sophisticated form of error handling is controlled by the 'X' bit in the SWI number which was mentioned earlier. When this bit is set the SWI will return to the calling application with the 'V' flag in the appropriate state (as above). However, it is frequently convenient to have a general-purpose error handler within an application (for example, the BASIC statement ON ERROR) which deals with all errors in a consistent way. By

issuing the SWI instruction with the 'X' bit clear, error control is passed by the Operating System to the currently defined 'error handler', which takes appropriate action. SWIs which behave in this way are known as 'error generating'. The error handler is established through the use of an Operating System 'vector', and this and other vectors are discussed in the next section.

The SWI naming system allows the state of the 'X' bit to be controlled by prefixing the SWI name with a capital 'X'. The default for SWI error control is the error generating state, ensuring that errors are flagged by the current error handler automatically. Error-returning SWIs use the 'X' prefix, so the error-returning form of the name conversion SWI mentioned above is:

```
XOS_SWINumberToString
```

Note that it is vital to remember to include software to deal with errors if the error-returning form is being used.

Error Handling – Numbering

In much the same way as SWIs are uniquely numbered, so the errors which SWIs produce are numbered in a consistent way. The error number field in an error block is 32 bits long and the bits are used as follows:

Bit 31	If set, indicates a serious error from which returning is impossible
Bit 30	Defined to be clear, but may be set to indicate internal errors
Bits 29-24	Reserved
Bits 28-3	Error generator identification field (see below)
Bits 7-0	Error number (0-&FF)

The middle two bytes of the error number form a field which identifies the particular piece of software responsible for generating the error. This is similar to the SWI 'chunk' field, but note that it has no numerical relationship with the chunk number whatsoever. Of the 65535 available error generator indentification fields, a small number have been allocated to existing software such as the Operating System. Just as for SWI chunks, software authors should apply to Acorn for an error generator field of their own.

Error Generation

As well as SWIs being able to generate errors, it is also possible for programs to generate errors of their own in order to signal unusual conditions. This is achieved by issuing the SWI OS_GenerateError with R0 pointing to an error block of the usual format, eg:

```
130 ADR R0,EscapeError ;Point to the error block
140 SWI "OS_GenerateError" ;Generate an error
```

A good example of the need for this facility is the detection and handling of the 'ESCAPE' key being pressed. All good software regularly checks to see whether the user has pressed the ESCAPE key, which is usually an indication that the user wishes to abandon or cancel the current operation.

The depression of the ESCAPE key can be tested for by issuing another SWI – OS_ReadEscapeState – which returns with the carry flag 'C' set if the key has been pressed. Listing 4.2 illustrates this. OS_ReadEscapeState is repeatedly called until the ESCAPE key has been pressed (ie, until the carry is set on return). Then, R0 is loaded with the address of the errorblock 'EscapeError' and the new error is generated by calling the appropriate SWI OS_GenerateError. Note that there is *no* return instruction following the call to OS_GenerateError: the reason for this is that your code will never be returned to from this SWI!

Listings

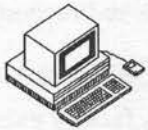
```
 10 REM >List4/1
 20 REM by Nicholas van Someren
 30 REM Archimedes OS: A Dabhand Guide
 40 REM (c) Copyright AvS and NvS 1988
 50 :
 60 DIM buffer% 127
 70 VDU 14
 80 :
 90 REM Loop through the first 256 SWIs, find
100 REM their names and print them.
110 :
120 FOR swi%=0 TO 255
130 SYS "OS_SWINumberToString",swi%,buffer%,127
140 SYS "OS_Write0",buffer%:PRINT
150 NEXT
160 VDU 15
170 END
```

Listing 4.1. Demonstrating OS_SWINumberToString

```
 10  REM List>4/2
 20  REM by Nicholas van Someren
 30  REM Archimedes OS : A Dabhand Guide
 40  REM (c) Copyright AvS and NvS 1988
 50  :
 60  DIM code% 100
 70  FOR pass%=0 TO 3 STEP 3
 80  P%=code%
 90  [OPT pass%
100  .loop
110  SWI "OS_ReadEscapeState" ;Wait until Escape pressed
120  BCC loop
130  ADR R0,EscapeError ;Point to the error block
140  SWI "OS_GenerateError" ;Generate an error
150  :
160  .EscapeError
170  EQUD 17
180  EQUS "I knew you were going to press Escape"
190  EQUB 0
200  ALIGN
210  ]:NEXT pass%
220  :
230  CALL loop
240  END
```

Listing 4.2. Demonstrating OS_ReadEscapeState.

5 : Command Line Interpreter



The Operating System Command Line Interpreter (OS_CLI) allows you to use many of the functions provided by SWIs without having to go to the effort of writing an assembler program. Obviously this is more useful as far as the user is concerned, but the penalty paid is one of speed – it takes time for the OS to decode the command and its parameters before the the appropriate SWIs are actually invoked.

Aside from the commands that the OS understands, modules may also add to the available OS_CLI commands; this is how the different filing systems provide their own device-specific commands. You too can add OS_CLI commands provided by your own modules – this is discussed in the chapter on modules (Chapter 11).

If you are familiar with the BBC Microcomputer MOS you will recognise the style of Operating System commands, which are preceded by a '*'. Many of the commands understood by the BBC MOS are also understood by the Archimedes OS, providing some consistency and making life easier for users familiar with the BBC. Most interactive applications written for Acorn computers provide some means for entering Operating System commands; for example, ARM BASIC simply passes on any line that starts with an asterisk.

To cater for the entry of * commands without having an application running the OS Supervisor (language number 0) allows the direct entry of command lines, displaying an asterisk as a prompt and allowing you to miss out the initial asterisk when entering the command.

OS_CLI Command Syntax

A number of other special characters may be included in OS_CLI commands to affect the way in which they are processed. These are detailed below.

Leading Spaces and Asterisks

Because it is very easy to type an extra asterisk by mistake, OS_CLI ignores all asterisks which precede commands. Similarly, all leading spaces are also ignored so as to allow some flexibility of syntax. Unfortunately, there

is some inconsistency in the way spaces are used, particularly with regard to 'redirection' which is discussed below.

Comments

A facility exists to tell OS_CLI to ignore a command line so that it may be used as a comment: this is achieved by putting a vertical bar '|' at the start of the line (ie, after any leading asterisks or spaces). One reason you might wish to do this is to allow an 'EXEC' file to contain both commands and comments which describe the commands, or to display instructions for the user without OS_CLI trying to interpret them.

Command Decoding Extensions

Several characters have reserved meanings within command lines. For example, the forward oblique or 'slash' ('/') is used as an abbreviation for *RUN, so the following are identical:

```
*RUN FileToBeRun
*/FileToBeRun
```

The colon (':') is used to follow filing system names to allow a temporary change of filing system. This makes it possible to perform some quick action on another filing system without having to explicitly change between the two. Hence, the following command sequences are equivalent when the ADFS is the current filing system:

```
*NET
*INFO  $.Alex.Book.OS_CLI
*ADFS
```

and, rather more quickly:

```
*INFO  NET:$.Alex.Book.OS_CLI
```

File Redirection

A feature of the Operating System which will be new to BBC MOS users is 'file redirection'. This allows characters to be read from files rather than the keyboard, and characters to be sent to a file instead of the screen. Thus, the effect of redirection is similar to having an EXEC or SPOOL file opened automatically before the command is executed.

The 'greater than' ('>') and 'less than' ('<') symbols, sometimes known as 'angle brackets' are used to represent file redirection. To allow OS_CLI to decode this feature braces ('{','}') must also be used before and after the

redirection command, and they must be preceded and followed by exactly one space.

As an example, here is a command which catalogues the current directory and places the resultant output into the file 'CurrentCat' (don't forget to follow the spacing exactly):

```
*CAT { > CurrentCat }
```

Note that the file CurrentCat will be created if it does not already exist, but its contents will be overwritten if it does. You may display the contents of the file CurrentCat (which will look exactly like a normal catalogue) with:

```
*TYPE CurrentCat
```

Conversely, the following command would run the BASIC program 'TestProg' with all its input taken from the file 'ProgInput':

```
*BASIC -quit { < ProgInput } TestProg
```

Finally, it it possible to append any output to a file (rather than overwriting the file if it exists) using two 'greater than' symbols, thus:

```
*EX { >> CurrentCat }
```

Once again, the *TYPE command may be used to display the new version of this file.

These redirection commands are a very powerful way of creating automatic command sequences, since it is possible to override the normal input and output of any command in this way.

Command Line Aliases

OS_CLI commands may be assigned 'aliases' allowing more than one name to be given to commands, this makes it possible to assign names to commands which you personally prefer as alternatives for the built-in names.

OS_CLI supports aliases of the form 'Alias$command', where 'command' is the name of the alias we wish to add. The OS_CLI command *SET is used to set up the alias, and any parameters that follow the command may be passed on using the pseudo-variables %0 to %9.

A good example is the case of the BASIC command MODE. Most BBC Micro users will have, at one time or another, mistakenly typed something like '*MODE 3'. Using aliases we can add a *MODE command to make this possible:

```
*SET Alias$MODE ECHO <22> <%0>
```

(The ECHO command will be discussed shortly, but its meaning is fairly self-evident.)

The effect of this command is to allow us to type *MODE mode_number, whereupon OS_CLI will expand the alias to issue:

```
VDU 22,mode_number
```

and thereby achieve the desired effect.

The same principle can be applied to get around another common error. Users frequently type '*>' instead of '*.' because they have not released Shift quickly enough. To handle this we can use:

```
*SET Alias$> CAT
```

Finally, besides the parameter pseudo-variables %0 to %9 we may also use the pseudo-variable '%*x' to pass on all of the rest of the command line after parameter number 'x'. This is particularly useful when we are simply providing an alias for an existing command. As an example, consider the command 'cc' which is used to compile a program written in the language 'C'. If we wanted an alias 'compile' for this command, we would simply need to type:

```
*SET Alias$compile cc %*0
```

and, although the command 'cc' takes a variable number of parameters, our alias 'compile' will always work.

OS_CLI Commands

The rest of this chapter is devoted to the commands that OS_CLI recognises on its own. A large number of other '*' commands are provided by the filing systems and by other software modules; these are described in the appropriate chapters elsewhere in this book.

When discussing the syntax of these commands we use the following notation for parameters:

<name>	a parameter name to be filled in as appropriate (without the angle-brackets)
{<name>}	an optional parameter
x | y	two parameters, either of which but not both may be used

*CONFIGURE

Syntax:

```
*CONFIGURE {<parameter 1> {<parameter 2>}}
```

The *CONFIGURE command displays or sets the various machine options which are stored in non-volatile memory. These are used to initialise the machine on power-up or hard break (CTRL-Reset). It is important to note that configuration settings do not take effect immediately, so it is usually necessary to issue the appropriate *FX command (or similar) if you want to change a setting right away.

When the command is issued with no parameters, all the available configuration options are listed.

When used with one parameter, the named option is configured.

When used with two (or more) parameters, the option named <parameter 1> is configured to the value of <parameter 2> etc.

Where numerical values are required, several forms may be used:

nnn	A decimal number
&nn	A hexadecimal number
base_nnn	The number nnn in number-base 'base', eg, 2_1111 is the same as &F which is the same as (decimal) 15.

A huge number of possible configuration parameters exist, more than it makes sense to cover in a book such as this. For more information refer to the relevant section of the *User Guide* or the *Programmer's Reference Manual*.

*ECHO

Syntax:

```
*ECHO <string>
```

This command translates the string provided as the parameter using the Operating System routine OS_GSTrans and then prints out the result. OS_GSTrans understands the split-bar format for control codes (eg, |B for turning on the printer) as well as aliases and other Operating System variables. It is also possible to include characters by putting their ASCII code in angle brackets, for example:

```
*ECHO <22> <0>
```

has the same effect as MODE 0 in BASIC. Similarly:

```
*ECHO <Alias$.>
```

displays the alias defined for the command *. (which is equivalent to *CAT unless the alias has been altered).

*ECHO is not particularly useful on its own, but as we saw earlier it may be used to define new OS_CLI commands which control the VDU.

*ERROR

Syntax:

```
*ERROR <error number> <error message>
```

This command generates an error whose number and associated message are supplied. This is most useful for raising errors to do with new OS_CLI commands, eg:

```
*ERROR 123 You can't do that!
```

*EVAL

Syntax:

```
*EVAL <expression string>
```

This command evaluates the expression supplied and displays the result and is the OS_CLI version of the SWI call OS_EvaluateExpression, which is documented in the section on conversion facilities. It allows string manipulation and simple arithmetic involving Operating System variables and integers, eg:

```
*EVAL <alex>+1024
```

*FX

Syntax:

```
*FX <parameter 1> {{,}<parameter 2> {{,}<parameter 3>}}
```

*FX has the same effect as it does in the BBC MOS, which is to say it calls the appropriate OS_Byte routine with the parameters supplied. For example:

```
*FX 5,4
```

sets printer output to be directed to the Econet printer.

Either commas or spaces may be used to separate the parameters, whose number varies according to the nature of the routine.

*GO

Syntax:

```
*GO {<parameter 1>} {<argument list>}
```

*GO is used to begin execution of machine code at the address specified by <parameter 1>, or at address &8000 if it is not supplied. In either case, the argument list is passed to the called program using registers. This allows an application program to be loaded at a fixed address and then called later, rather than using *RUN or similar.

*GOS

Syntax:

```
*GOS
```

This command, an abbreviation for 'Go Supervisor', enters the Operating System Supervisor which then displays its familiar '*' prompt.

*HELP

Syntax:

```
*HELP {<keywords>}
```

The *HELP command has the same effect as the BBC MOS command of the same name, ie, it displays help text on topics whose keywords are provided.

Because Archimedes computers have a large amount of ROM there is at least a line or two of help information on every resident command, so you can expect a useful result from this command for almost every eventuality. Try typing:

```
*HELP COMMANDS
```

this will produce a list of the modules currently available, each of which is followed by a list of keywords on which more help is available. For example:

```
*HELP MOUNT DISMOUNT
```

lists a line or two of help on the ADFS commands *MOUNT and *DISMOUNT.

It is possible to abbreviate keywords by ending them with a full stop. This produces help on all the keywords beginning with the given sequence of letters, so:

```
*HELP S.
```

will produce help on all the available commands that begin with an 'S'.

*IF

Syntax:

```
*IF <expression> THEN <command 1> {ELSE <command 2>}
```

This very powerful statement allows commands to be executed conditionally on the result of <expression>. If <expression> is 'true', which is to say yields a non-zero result, then <command 1> is executed. Otherwise, where the 'ELSE' part of the command has been included, <command 2> is executed.

The expression is evaluated using the SWI OS_EvaluateExpression in the same way as *EVAL, so it may include system variables, integer arithmetic or string manipulation if desired. For example, if there were two directories – one called "1988" and one called "1989" then we could enter the appropriate one automatically by using:

```
*IF <Sys$Year>="1988" THEN DIR 1988 ELSE DIR 1989
```

*IGNORE

Syntax:

```
*IGNORE {<ASCII character code>}
```

The *IGNORE command has the same effect as *FX 6 and is the immediate form of the *CONFIGURE IGNORE command; ie, it sets the character which will not be sent to the printer. The main use of this is for stripping unwanted line feeds (ASCII 10) from the printer stream to avoid extra blank lines. For example:

```
*IGNORE 10
```

prevents line feeds from being sent to the printer. When no character code is supplied, all characters are sent to the printer.

*KEY

Syntax:

```
*KEY <key number> {<key definition>}
```

This command is the same as its BBC MOS counterpart. It allows a string to be assigned to one of the sixteen function keys so that each time the key is pressed subsequent input is provided by the defined string. For example:

```
*KEY 1 *GOS|M*ADFS|M*DISMOUNT|M*BYE|M
```

sets up function key 'f1' to enter the OS supervisor (*GOS), select the ADFS (*ADFS), dismount the currently selected disc (*DISMOUNT) and park the disc heads if a hard disc is selected (*BYE). Notice the use of GS_Trans style control codes (in this case |M, representing RETURN) which are decoded before being stored as the key's definition.

The sixteen keys comprise the 'PRINT' key (zero), the row of keys labelled f1 to f12, the 'COPY' key (eleven) and the left, right, down and up arrow keys (twelve to fifteen respectively).

An Operating System variable called KEY$<key number> exists for each key. This may be assigned using *SET (see below).

*SET

Syntax:

```
*SET <variable name> <string>
```

*SET assigns the given string to the named Operating System variable, creating the variable if it doesn't already exist.

Variables can be either of type 'number' or type 'string', with obvious effects on expression evaluation (you can't multiply strings!). The *SET command is used just for strings – *SETEVAL is used for numbers (see below).

As an example of *SET:

```
*SET Silly "This is a silly message"
```

creates and assigns the string to the variable 'Silly', which may then be displayed with *ECHO, thus:

```
*ECHO <Silly>
```

Notice the use of angle-brackets to force *ECHO to evaluate the variable, rather than just printing the word 'Silly'.

The OS provides a number of built-in variables which already exist and may be altered by the user, though never deleted. They include the following:

Variable name	Contents
Sys$Time	Current time in the style 23:59:59
Sys$Date	Current date in the style Fri, 31 December
Sys$Year	Current year in the style 1988

So, to change the time we might say:

```
*SET Sys$Date Fri, 25 December
*SET Sys$Year 1987
```

In addition, other parts of the Operating System or other modules may add further variables of their own, for example:

Variable name	Contents
Cli$Prompt	Command line prompt string (default is '*')
Key$1	String associated with function key f1

*SETEVAL

Syntax:

```
*SETEVAL <variable name> <expression>
```

This command allows numeric variables to be assigned. It evaluates the expression using OS_EvaluateExpression and then assigns it to the named variable, creating the variable if necessary. For example:

```
*SET Control 0
*SETEVAL Control Control+1
```

*SETMACRO

Syntax:

```
*SETMACRO <variable name> <expression>
```

*SETMACRO is almost the same as *SETEVAL, but the evaluation of the expression is deferred until each time the variable is accessed. The assignment can therefore change each time. The classic use of this is to have an OS_CLI prompt which is the time, thus:

```
*SETMACRO CLI$Prompt <Sys$Time>
```

which has got to be the ultimate aid for clock-watchers!

*SHADOW

Syntax:

```
*SHADOW {<value>}
```

This command is provided explicitly to allow compatibility with the BBC MOS (on machines from the BBC B+ onwards). Values of 1 and 2 force subsequent mode changes to use screen memory banks 1 and 2. Bank 2 is used if no value is supplied.

*SHOW

Syntax:

```
*SHOW {<variable name>{*}}
```

*SHOW displays the name, type and current value of system variables. When no variable name is supplied all variables are displayed; otherwise a specific variable is displayed, eg:

```
*SHOW Cli$Prompt
```

which displays a result of the style:

```
CLI$Prompt : type String, value : Yes, Master?
```

By appending an asterisk to a partial variable name a wildcard effect is introduced, so:

```
*SHOW Sys$*
```

will display all of the system variables. Similarly, the following will display the current assignments of all the function keys:

```
*SHOW Key$*
```

*STATUS

Syntax:

```
*STATUS {<configuration option>}
```

This command allows you to examine the current setting of one or all of the configuration options set with *CONFIGURE. When no parameter is supplied the entire list will be displayed. Otherwise, if a valid option name is supplied, its current setting is displayed.

For example, to display the configured screen mode:

```
*STATUS MODE
```

or to display the configured international language setting:

```
*STATUS COUNTRY
```

Don't forget that the settings displayed by *STATUS are those that will come into effect *next* time the machine is powered-up or reset – they are not necessarily prevailing at the time the *STATUS command is issued.

*TIME

Syntax:

```
*TIME
```

This command displays the time and date in the format defined by the system variable Sys$DateFormat. The default is of the style:

Fri,31 Dec 1999.23:59:59

though this format can be changed by altering Sys$DateFormat – consult the SWI OS_ConvertStandardDateAndTime in the section on numeric conversion facilities for information on formatting.

*TV

Syntax:

```
*TV {<offset> {{,}<interlace>}}
```

This is the immediate form of the TV configuration option. Note that its settings do not take effect until the next change of display mode.

The offset parameter allows the whole display picture to be moved up and down by a number of character lines: 1 means move up by one line, 255 means move down by one line, etc.

The interlace parameter controls whether an interlaced picture is produced: a zero switches interlace on, a one switches it off. If this parameter is not supplied then the interlace state remains unaltered.

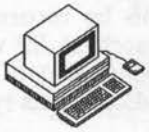
*UNSET

Syntax:

```
*UNSET <variable name>{*}
```

*UNSET deletes one or more non-system variables. If an asterisk is used then all variables which begin with the characters supplied will be deleted. You cannot delete a system variable, although no error will be reported if you try to do so.

6 : OS_CLI Related SWIs



This chapter details the SWIs which relate to the CLI, specifically OS_CLI SWI itself and the two SWIs which deal with Operating System variables as used by *SET and *SHOW.

OS_CLI (SWI &05)
Command Line Interpreter

On entry, R0 should point to a command string (without any leading asterisks) terminated by any of zero, line feed (ASCII 10) or carriage return (ASCII 13). The string will be processed as an Operating System command and its effects and the results returned are dependent on the command.

OS_ReadVarVal (SWI &23)
Read the Value of an OS Variable

This call allows the existence or the type and value of one or more OS variables to be established. On entry, R0 should point to a zero terminated string containing the name of the variable(s), optionally including the wildcards '*' to match zero or more characters and '#' to match exactly one character. R1 should point to a buffer for the SWI to use, with R2 containing the maximum size of that buffer. A special case is provided by setting bit 31 of R2 which simply determines whether the specified variable(s) exist by returning with R2=0 if not.

On the first call to this SWI, R3 should contain zero. If wildcard matches are expected then subsequent calls should be made with the previous contents of R3 preserved, its value being updated automatically by the SWI. The 'XOS' form of this SWI should be used where wildcard matches are expected in order to avoid the 'no more matches' error which will be generated after the last match is read.

R4 should contain three if the result should be expanded by OS_GSTrans before returning (see next page). Other values are ignored.

After the first call to this SWI, the type and value of the specified variable will be returned (or those of the first match in the case of wildcard searches). R2 will contain the number of bytes which were read from the string, R3 will point to the variable's value and R4 will contain a number indicating its type.

The table below summarises the possible OS variable types:

R4	Type
0	String
1	Signed 32-bit integer
2	Macro

If R4 does not contain three on entry, the value of the variable is returned either as an integer or as a string. If R4 does contain three then integers are converted to signed strings and GS_Trans is called to expand any variable names in macros. In the latter case, R3 points to the start of the string.

OS_SetVarVal (SWI &24)
Create/Assign to/Delete an OS Variable

This call allows OS variables to be created, deleted or have their value altered. Note that not all of the possible types which may be created can be read with the previous SWI.

On entry R0 should point to the name of the variable terminated by a space or control character. R1 should point to the value to which the variable should be set, its format dependent upon the type of the variable. R2 should contain the length of the value (but see below). R3 should be set to zero on the first call and will be updated by the SWI to deal with wildcards when assigning or deleting. Finally, R4 should contain the type of the variable from the table below:

R4	Treatment of value
0	Value is OS_GSTransed before assignment
1	Value is taken as an integer
2	Value is taken as a string
3	Value is passed through OS_EvaluateExpression
16	Value is a piece of code to execute (see below)

To delete a variable the top bit of R2 should be set for types 0-3 and R2 should contain 16 for type 16.

When the call returns, R3 will be updated and should be preserved for the next call if wildcards are being employed. R4 returns the type that was

created if OS_EvaluateExpression was used (since it might be either a string or integer).

If the 'code' type is used you can supply your own code to deal with read and write operations. In this case, R1 should point to the header of the code and R2 should contain its length. The code must be preceded by a header which consists of two branches which call the write routine and the read routine respectively. The code is called in Supervisor Mode so R14 should be preserved on the system stack before any SWIs are used.

The write entry point is called with R1 pointing to the value to be written and R2 containing its length. The read entry point is called with no entry parameters and your code should return a pointer to the value in R0 and its length in R2. A number of errors may be generated by OS_SetVarVal according to various parameter or syntax errors in its use.

Marvin

There follows an extensively annotated example (affectionately known as 'Marvin') which illustrates the use of the SWI OS_SetVarVal. It uses some devious code to randomly change the value of Arthur variable Cli$Prompt – the variable that contains the string printed when Arthur awaits a command. The program assembles a piece of code called "Marvin" and *MARVIN will load and execute this code. To see its results type QUIT. And when you've had enough, press <CTRL-BREAK>!

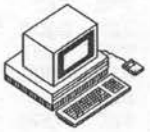
```
 10 REM >List6/1
 20 REM MarvinSrce
 30 REM by Nicholas van Someren
 40 REM Archimedes OS : A Dabhand Guide
 50 REM (C) Copyright AvS and NvS 1988
 60 DIM code% 1000
 70 FOR pass%=0 TO 3 STEP 3
 80 P%=code%
 90 :
100 [OPT pass%
110 .start
120 SWI "OS_ReadMonotonicTime" ;Get random number seed
130 STR R0,seed                ;Save it
140 ADR R0,varname             ;Get variable name's address
150 ADR R1,code                ; and address of our code
160 MOV R2,#endcode-code       ; and length of our code
170 MOV R3,#0                  ;Use first match of name
180 MOV R4,#&10                ;'Marvin' is of type 'code'
190 SWI "OS_SetVarVal"         ;Create variable 'Marvin'
200 ADR R0,cliname             ;Get 'Cli$Prompt' string address
210 ADR R1,clistr              ; and address of value string
220 MOV R2,#cliend-clistr      ; and length of value string
```

```
230 MOV R3,#0                       ;Use first match
240 MOV R4,#2                       ;'Cli$Prompt' is of type 'string"
250 SWI "OS_SetVarVal"              ;Create new 'Cli$Prompt'
260 MOV PC,R14 ;Return
270 :
280 .code                           ;The start of 'Marvin' code
290 B writecode                     ;Branch to write code
300 .readcode                       ;Start of read code
310 LDR R1,seed                     ;Fetch the random seed
320 EORS R1,R1,R1,ROR #13           ;Shift it and 'EOR' with itself
330 LDR R2,constant                 ;Get a constant
340 ADDCS R1,R1,R2                  ;Add it in when carry set by ROR
350 STR R1,seed                     ;Store this back as next seed
360 AND R1,R1,#&3C                  ;Mask off all but bits 2-5
370 ADR R0,strings                  ;Get address of pointer table
380 ADD R1,R1,R0                    ;Add in random offset
390 LDR R0,[R1],#4                  ;Read pointer, post incremented
400 LDR R2,[R1]                     ;Read next pointer
410 SUB R2,R2,R0                    ;Store the difference in R2
420 ADR R1,ss                       ;Get absolute address of strings
430 ADD R0,R0,R1                    ;Form real address of message
440 .writecode                      ;Write code is put here
450 MOV PC,R14 ;Return
460 :
470 .seed                           ;Space for a random seed
480 EQUD 12345678
490 :
500 .constant                       ;A prime constant
510 EQUD 555557
520 :
530 .strings                        ;The string offset table
540 EQUD s0-ss
550 EQUD s1-ss
560 EQUD s2-ss
570 EQUD s3-ss
580 EQUD s4-ss
590 EQUD s5-ss
600 EQUD s6-ss
610 EQUD s7-ss
620 EQUD s8-ss
630 EQUD s9-ss
640 EQUD sA-ss
650 EQUD sB-ss
660 EQUD sC-ss
670 EQUD sD-ss
680 EQUD sE-ss
690 EQUD sF-ss
700 EQUD s10-ss
710 :
720 .ss    ;The strings themselves
730 .s0
740 EQUS "That was an amazing command."
750 .s1
```

```
 760 EQUS "Four Mips, and you ask me to do that."
 770 .s2
 780 EQUS "I'm NOT doing that again."
 790 .s3
 800 EQUS "I've got a pain in all the diodes down my side."
 810 .s4
 820 EQUS "Brain the size of a planet, and I'm doing this."
 830 .s5
 840 EQUS "You'll ask me to pick up a piece of paper next."
 850 .s6
 860 EQUS "I hate humans."
 870 .s7
 880 EQUS "Why don't you do something mindless instead?"
 890 .s8
 900 EQUS "Don't hit the keys so hard."
 910 .s9
 920 EQUS "Shall I put my head in a bucket of water?"
 930 .sA
 940 EQUS "Is there any point to this?"
 950 .sB
 960 EQUS "I could blow myself up if you wanted me to."
 970 .sC
 980 EQUS "Don't talk to me about life."
 990 .sD
1000 EQUS "I'm not impressed by your typing."
1010 .sE
1020 EQUS "You're really very stupid, compared to me."
1030 .sF
1040 EQUS "You can't realise how dull this is."
1050 .s10
1060 ALIGN
1070 .endstring ;The end of the strings
1080 .endcode   ;The end of the code
1090 .varname   ;The name of the variable
1100 EQUS "Marvin "
1110 ALIGN
1120 .cliname   ;The prompt variable name
1130 EQUS "Cli$Prompt "
1140 ALIGN
1150 .clistr    ;The prompt variable's value
1160 EQUS "<Marvin> M J*"
1170 EQUB 0
1180 .cliend    ;End of the prompt variable
1190 ]:NEXT
1200 :
1210 REM Save the code and make it a utility
1220 :
1230 OSCLI"SAVE Marvin "+STR$~code%+" "+STR$~P%
1240 OSCLI"SETTYPE Marvin FFC"
1250 PRINT"Code assembled and saved as 'Marvin'."
1260 END
```

Listing 6.1. Marvin.

7 : Filing Systems



Introduction

Filing systems are the parts of the Operating System which are responsible for the storage and retrieval of large amounts of information collectively known as 'files'. Each different storage medium has its own filing system to deal with the housekeeping and hardware control neccessary for the storage device concerned. Because many of the operations that filing systems perform are the same for all devices, the Operating System divides the management of files into two sections: a generic part, concerned with all filing systems, and a specific part which deals with a specific kind of device. The generic part is known as the 'FileSwitch' and it controls the interface between the user and a particular filing system. This approach has many advantages, not least that it keeps the 'user interface' (syntax of commands) consistent and further, it reduces the amount of software required to implement a new filing system.

The filing systems which are most intensively used on Archimedes computers are the Advanced Disc Filing System (ADFS) and the Econet local area network filing system (NFS). An introduction to the ADFS is included in the User Guide which is part of the Archimedes package – if you are not already familiar with the basic operation of the ADFS then you should read the chapter on filing systems in the User Guide before reading the rest of this chapter.

The OS also provides some 'device' filing systems which allow the keyboard, screen and printer to be treated as filing systems in themselves. This mechanism makes input/output more consistent across all of the standard peripherals. We shall examine these peripheral devices in more detail later in this chapter. While the Arthur OS doesn't support a 'RAM filing system' RISC OS does and this will be a major time-saver for floppy discs users. No doubt many third-parties will produce filing systems for their own devices which will hopefully keep to the standards set by existing Acorn software.

Naming of Filing Systems, Files and Directories

Most filing systems consider data as an arbitrary collection of bytes referred to by name – this is the definition of a 'file'. Every file has a name, the 'filename', which uniquely identifies it within the current area of work. Filing systems do not themselves make any distinction between the kind of data stored in a file; it is up to the Operating System, the user and the application to determine whether the data in a given file is of a suitable format.

The naming of files must obey certain rules in order to allow filing systems to operate in a consistent way. In particular, filenames should ideally be composed exclusively of letters, digits or underline symbols – they should not contain any punctuation symbols because most of these are reserved for special meanings.

Both the ADFS and NFS support filenames of up to ten characters, with no distinction being made between upper and lower case letters. You should be extremely cautious when using foreign language (or 'top-bit-set') characters in filenames because the support for them is very limited in current software and they will have nasty side-effects if files are transferred to older systems, particularly BBC Micro implementations of the ADFS.

The punctuation symbols shown below have special meanings that prevent them from being used in filenames – if you use them by mistake you can expect error messages to result:

. : - * # @ $ & \ ^ %

Directories

Most filing systems provide a means of grouping files together in clumps known as 'directories'. It is entirely up to the user how directories are used, but it is common to group related files in the same directory to make them easier to find. An important restriction is that all the filenames within a given directory must be unique, otherwise it would be impossible to be sure which file was being referred to. However, directories may contain other directories (called 'sub-directories') and so on, leading to a 'hierarchical' arrangement of nested directories.

The top of a directory hierarchy is known as the 'root' and is represented in Acorn filing systems by the dollar symbol '$'. Each disc device, whether it is floppy or hard, has a root whose 'real' directory name is the name of the

disc. Thus the disc name (preceded by a colon ':') can usually be used interchangeably with the dollar symbol to refer to the root directory.

At any given time one of the directories in the hierarchy will be in use – this is known as the Currently Selected Directory (CSD) and is represented by the ampersat symbol '@'. Files within the CSD may be referred to by their name alone; files in other directories must be referred to by a name sequence which uniquely identifies them and is known as their 'pathname'. The pathname is formed by stringing together the appropriate sequence of directory names connected by full stops. Thus, a file called 'Fred' in directory 'Alex' which is in turn a subdirectory of the root has the full pathname '$.Alex.Fred'.

The other reserved punctuation symbols noted earlier have special meanings when used in pathnames; these are:

. Joins directory/file names in a pathname, eg:
Alex.Fred

: Precedes a disc name (equivalent to $), eg:
:MyDisc.Alex

Note that the colon is also used to follow a filing system name; see later.

$ Represents the root directory of the disc, eg, $.Alex
& Represents the User Root Directory (URD)
@ Represents the Currently Selected Directory (CSD)
^ Represents the 'parent' directory
% Represents the Currently Selected Library (CSL)
\ Represents the Previously Selected Directory (PSD)

The URD and CSL have not been mentioned before, but are discussed later in this chapter. The meaning of the 'parent' directory is fairly self-evident, being the directory which contains the directory so far specified in a pathname. The use of the caret symbol '^' is intended to convey movement 'up' the hierarchy, with the root imagined as being at the top.

Files on Different Filing Systems

A powerful extension to pathnames allows them to specify that a file is to be found on a particular filing system; this allows pathname strings to be composed for any file on any filing system, but without the burden of needing to change filing systems back and forth manually.

A filing system name is introduced by following it with ":". The logic of this is that the next thing usually specified is a disc name (though this is

optional), so the consistency of the naming system is preserved. For example:

```
NET:$.Alex.Fred
```

refers to the file 'Fred' in the directory 'Alex', which is a sub-directory of the root of the currently-selected network file server. Similarly:

```
ADFS::BigDisc.Programs.BASIC.Pretty
```

refers to the file 'Pretty' which is in the sub-directory 'Programs.BASIC' on the disc entitled 'BigDisc'. Note that here two colons are required, one to follow the filing system name and the other to precede the disc name.

For BBC MOS compatibility reasons the old style of enclosing filing system names in hyphens, eg, –NET–, is still tolerated by the OS but its use should be avoided wherever possible.

Device Filing Systems

Another very useful aspect of filing system naming is that the keyboard, display and printer may be treated as filing systems in themselves for the purpose of byte-oriented operations. It is not sensible to try to load or save programs to the device filing systems!

Six different device filing systems are currently supported (seven if we include null:). They are listed below along with a brief note on their functionality:

Device Name	Functions Available
kbd:	Input only, returned by OS_ReadLine
rawkbd:	Input only, returned by OS_ReadC
vdu:	Output only, processed by GS_Read then sent to OS_WriteC
rawvdu:	Output only, issued through OS_WriteC
printer:	Output only, issued direct to current printer
null:	
Input/Output:	Input returns End Of File, Output is discarded

The 'raw' forms of the keyboard and VDU devices deal with characters exactly as they are encountered, whilst the 'cooked' forms apply the specified pre-processing (mainly to cater for escape sequences using the vertical bar '|') before passing the data on. Note that the way to generate an End Of File condition on the keyboard is to type <CTRL-D> and press RETURN. In case you are wondering, this mechanism is borrowed from the Unix Operating System, perhaps a hint of things to come.

Quite aside from the fact that the device filing systems are just plain useful, their existence preserves the consistency of the the OS device naming syntax through nearly all the devices available on standard Archimedes computers (the notable exception being the RS423 port). If you consider the potential difficulties of signifying the end of file on the RS423 device it is not hard to see why the authors of the OS chose not to implement it right away.

It may have occurred to you that the ability to treat the VDU and keyboard as filing systems means that several of the standard Operating System utilities are really redundant; for instance, *BUILD and *TYPE. However, compatibility requires them to be provided nevertheless, and this seems to be the main reason they survive.

Ancillary File Information
Load and Execute Addresses

As well as having a name, every file has two 32-bit fields associated with it which give the Operating System information about where it should be loaded in memory and where execution should begin. Not surprisingly, these are referred to as the 'load address' and 'execute address'. Load and execute addresses are only really necessary for machine code programs because the Operating System deals automatically with other kinds of files, as we shall see shortly.

If you use the *LOAD command to load a file into memory, the load address associated with the file is used by default. It is possible to override this by supplying a different load address as part of the *LOAD command, though this clearly requires the file to be relocatable if it is a piece of machine-code.

The *RUN<filename> command (and its synonyms *<filename> and */<filename>) allow machine code programs to be loaded and executed in one go – in this case the file is loaded at its load address and then executed starting at its execute address. The execute address must be within the program or the Operating System will generate an error. (By 'within the program' we mean it must be greater than or equal to the load address but less than the load address plus the length.)

If the top twelve bits of the load address are set, ie, if the first three hexadecimal digits are 'FFF', the file is treated in a special way. Such an address is outside the addressing range of the ARM so the OS uses this as a flag for storing file type information, allowing a number of useful automatic features to be provided.

File Types and Date Stamping

Where a file has a load address with the top twelve bits set, the remaining bits are used to keep three useful pieces of information: the file's type, and the date and time 'stamp', initially indicating precisely when it was created. Clearly it is very useful to have this information because it allows us to keep track of the 'age' of programs. Furthermore, the file's type allows us let the OS decide how to deal with a file when we issue non-specific commands such as *<filename>.

The format of the file type, date and time information stored in the load and execute address fields is as follows:

Load Address	FFFtttdd
Execute Address	dddddddd

Here, the file type information is represented by the three hexadecimal digits shown as 'ttt', and the date and time by the ten hexadecimal digits 'ddddddddddd'. When a file is created or has its date stamp updated, the absolute time is stamped onto the file in the form of the number of centiseconds (hundredths of a second) since 00:00:00 on the 1st January 1900 – this is usually a pretty large number! Whilst such accuracy is not generally required, it is readily available from the real-time clock and internal counters in the Archimedes, so it is stamped in full for speed. The command *STAMP may be used to update the date and time stamp on a given file, and this is discussed later in this chapter.

The Operating System uses the file type to decide what action to take when requested to *RUN the file. Since a command of the form *<filename> is a synonym for *RUN, we can be lazy and type only the name of a file to save time. It then looks up the file type and decides what to do.

Operating System aliases may be defined for each possible file type – they allow the two cases of *LOADing and *RUNning the file to be determined separately. Each file type may have two such aliases:

Run$Type_ttt	for *RUN, */ or *<filename>
Load$Type_ttt	for *LOAD

Whenever either of these actions is applied to a file, the Operating System checks for a defined alias and uses it if possible. For example, the aliases for files whose type is 'BASIC program' (type &FFB) are:

Load$Type_FFB	BASIC –Load %*0
Run$Type_FFB	BASIC –Quit %*0

which cause the BASIC interpreter to be entered with the appropriate options set and the remainder of the command line passed on with %*0. The file type aliases may be set using the *SET command as is usual for OS variables, so where we have files for our own software (file type &ABC) we can establish them with:

```
*SET Load$Type_ABC MyCode %0
*SET Run$Type_ABC MyCode %0
```

You can use *SHOW to display the current settings for file types. Here is a list of the file types currently allocated by Acorn for OS1.2:

Type	Meaning
&FFF	ASCII text
&FFE	EXEC commands
&FFD	Data
&FFC	Transient program
&FFB	BASIC program
&FFA	Relocatable module code
&FF9	Sprite definition
&FF8	Application code
&FF7	BBC Micro font definition
&FF6	OS font manager font definition
&FEF	Desktop diary file
&FEE	Desktop notepad file
&FED	Palette definition
&FE0	Desktop accessory code

The file type range from &000 to &7FF is free for use by user programs, all other values being reserved by Acorn.

Libraries and Search Paths

When a command is given to the Operating System, it first establishes whether the command is supported by any of the software installed in the computer's ROM or RAM. If not, the OS attempts to find a file of the same name in the CSD and execute it. Clearly, if all our extensions to the Operating System had to be provided in one directory, the hierarchical directory system would be wasted.

In order that we may enter OS commands to run files which are not in the CSD, we need to be able to indicate where else the OS should search. This is achieved through the use of 'libraries'. A second directory, known as the Currently Selected Library (or CSL), may be nominated as the place for Arthur to search when it encounters an unrecognised command. The CSL is

represented by the percent character '%' in pathnames, and may be set using the *LIB command, which is followed by the pathname of the chosen library directory. Thus, one or more directories (though only one at a time) can be used to keep a 'library' of command files. To allow us to override this, whenever a directory name is included explicitly in the command then other searches do not take place.

The Operating System takes this idea a stage further and provides OS variables which allow multiple search paths to be set up.

Two different classes of search path are available; one for commands, analogous to the library system described above, and one for other 'read' functions such as loading or opening files for input.

Two OS variables are provided to establish these search paths:

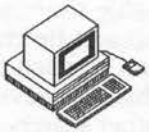
Run$Path	for execute operations
File$Path	for read operations

and these may be set using *SET in the usual way. Each may contain a sequence of directory pathnames, separated by commas and terminated by a full stop. The default settings for these variables are:

Run$Path	,%.
File$Path	

which is to say that commands files for execution are searched for first in the CSD, then in the CSL and that files to be read are only searched for in the CSD. What makes this facility particularly useful is that since pathnames may contain filing system names, the path strings can specify libraries on one or several different filing systems. Obviously, the amount of time spent dealing with the command increases as the number of items in the path strings do, particularly where multiple filing systems are in use, but frequently the impact of this is scarcely noticeable to the user.

8 : The FileSwitch Module



The FileSwitch is an Operating System module which makes OS_CLI '*' commands and SWIs available for general filing system control. By separating out the filing system functions which are common to all filing systems, the FileSwitch ensures the consistency of the user interface and makes the implementation of new filing systems easier by reducing the number of functions that need to be supported.

All FileSwitch commands are non-filing system specific, and they include commands for file and directory creation, deletion, cataloguing, examination, copying and so forth. Most of the FileSwitch OS_CLI commands have one or more equivalent SWIs which are used to call the selected filing system as appropriate.

FileSwitch Commands

The following few pages cover each of the FileSwitch commands in some detail – of course, *HELP will provide an online summary. In describing the syntax of these commands we use the word 'object' to mean either a file or a directory.

*ACCESS

Syntax:

```
*ACCESS <object name> [D] {L} {W} {R} {/} {W} {R}
```

The *ACCESS command allows the user to control the access attributes of a file, and allows limited control of directory attributes. It takes an 'object' name – the name of a file or directory (optionally including wildcards) – and a string of up to six attribute symbols. These attributes are applied to the object, replacing those that were previously present. The possible attributes are:

D Indicates that the object is a directory. This attribute may not be altered by the *ACCESS command but is included here for completeness.

L When present this attribute 'locks' an object, preventing it from being deleted, written to or overwritten except by specifically overriding the lock. This is primarily to prevent accidental damage to files, but directories always default to 'locked' so as to remind you to be cautious. Of course, the locked bit doesn't prevent the contents of directories being altered; only the deletion of the directory itself.

W The presence of this attribute allows writing to files. To successfully OPENOUT or OPENUP a file its access must include a 'W' and must not include an 'L'. This attribute has no relevance for directories.

R The presence of this attribute allows reading from files. To successfully OPENIN or OPENUP a file its access must include an 'R'. This attribute has no relevance for directories.

/ The oblique character separates the attributes for the two levels of security on Econet systems, known as 'private access' (before the oblique) and 'public access' (after it). Familiarity with the Econet is required to take full advantage of this attribute, which is not supported by the ADFS.

Example:

```
*ACCESS MyFile LR
```

Locks and prevents writing to 'MyFile'.

*APPEND

Syntax:

```
*APPEND <filename>
```

The *APPEND command is used to add text entered from the keyboard to the end of an existing file. It operates in the same way as *BUILD (see below), but requires that file already exists and then extends it. *APPEND is terminated by an ESCAPE condition.

Example:

```
*APPEND ExecFile
```

Allows text to be added to the end of 'ExecFile'.

*BUILD

Syntax:

```
*BUILD <filename>
```

*BUILD creates the named file (or, if it already exists, overwrites it if attributes permit) and then writes to the file all text entered from the keyboard. Its most common use is for the creation of *EXEC files (see below). *BUILD is terminated by an ESCAPE condition (that is usually, but not necessarily, by pressing the ESCAPE key). The default file type used by *BUILD is &FFD (Data), which must be changed to &FFE using *SETTYPE (see later) if the file is to be *EXECed automatically.

Example:

```
*BUILD !Boot
```

Creates and allows text to be entered into !Boot.

*CAT

Syntax:

```
*CAT {<directory>}
```

The *CAT command 'catalogues' (lists the contents of) the specified directory. If no directory pathname is given, the CSD is catalogued.

Examples:

```
*CAT
*CAT  $.Programs
```

Catalogues the CSD and the directory $.Programs.

*CDIR

Syntax:

```
*CDIR <directory> {<maximum size>}
```

The *CDIR command creates a new empty directory with the specified pathname, setting its attributes to the default of 'DL'. The NFS accepts the optional 'maximum size' parameter and ensures that the directory will be able to contain the specified number of objects. ADFS takes no notice of this parameter.

Example:

```
*CDIR  NewDir
```

Creates a new directory 'NewDir' in the CSD.

*CLOSE

Syntax:

```
*CLOSE
```

This command ensures that all file buffers in RAM are written out to the appropriate files and then closes all open files on the current filing system.

*COPY

Syntax:

```
*COPY <source spec.> <destination spec.> {<options>}
```

The *COPY command is a general-purpose file duplicating command of considerable power. It requires at least two parameters: a source specification and a destination specification (both of which may contain wildcards) and then copies all the specified objects from the source to the destination. Directory names may be used in both source and destination specifications, but are interpreted differently. When a wildcard is used in a directory name in the source, only the first match is used; when used in filenames a wildcard *must* appear in both source and destination, in which case the wildcard field is preserved. For example:

```
*COPY  $.Alex.*  Net:Alex.Safety.*
```

will copy all the files in '$.Alex' on the current filing system into the network directory 'Alex.Safety', preserving the names of the files as it does so. However, the following:

```
*COPY  *.Nick  NewFile
```

will only copy the file called 'Nick' from the first subdirectory of the CSD to the file 'NewFile'.

A range of options may be appended to the *COPY command:

C — Confirm. The user is asked to confirm that each file is to be copied.

D — Delete. Files are deleted after they have been copied.

F — Force. Destination files are overwritten if they already exist.

P — Prompt. The user is prompted for disc changes when copying between discs. This is for use with single disc drives.

Q — Quick. Allows application memory to be used during copying, thus speeding up lengthy copies considerably. However, programs in the application space will be destroyed, and the user will be returned to the OS Supervisor on completion of the command.

R — Recurse. Causes sub-directories to be copied as well, saving time when an entire directory structure needs copying. Note: it is unwise to recursively copy a directory into one of its sub-directories as this will result in an infinite loop which will fill up the disc!

V Verbose. Displays information about each file copied.

The Operating System pseudo-variable Copy$Options contains the default settings of these options. The default actions may be overridden by giving the desired attribute set after the command. For example:

```
*COPY This That ~RV
```

Any attribute not specified after the command assumes the state specified in the default Copy$Options variable. Specifying an option causes the corresponding action to be selected. Specifying an option prefixed by a '~' causes the option to be de-selected and the corresponding action is *not* taken.

In the above example, we specify that a Recursive copy (R) is *not* to take place but that we want to force Verbose mode (V) to be selected. The other options revert to their default settings.

*COUNT

Syntax:

```
*COUNT <file specification> {<options>}
```

The *COUNT command totals the size of the file(s) that match the specification (which may contain wildcards) and displays the results rounded to the nearest 1k. It also responds to the 'R' and 'V' options in the same way as *COPY, allowing multiple directories to be sized and the files involved to be displayed.

Example:

```
*COUNT $ R
```

Finds the total size of all files on a disc.

*CREATE

Syntax:

```
*CREATE <filename> {<size> {<execute address> {<load address>}}}
```

The *CREATE command reserves space for a file without actually writing any data into it. The optional size parameter sets the number of bytes reserved, zero being used by default. The execute and load addresses may be set, the load address defaulting to zero if not supplied. Where both addresses are omitted, the file type is set to &FFD (Data).

Example:

```
*CREATE BigFile 20000
```

Creates a new file 'BigFile' 128k long.

*DELETE

Syntax:

```
*DELETE <object name>
```

The *DELETE command deletes the named object from the catalogue so that the space it occupies may be re-used. An error message will be generated if the object does not exist or is locked or, in the case of directories, if the directory is not empty. Wildcards may be used in all fields of the pathname except the last and so, to delete several files at once, *WIPE should be used instead (see below).

Example:

```
*DELETE Junk
```

Deletes the file called 'Junk'.

*DIR

Syntax:

```
*DIR {<directory>}
```

This command changes the Currently Selected Directory (CSD). If no directory name is supplied the current directory is set to the user root directory (URD). If the command was processed successfully, the directory which was the CSD when the command was issued becomes the previously selected directory (PSD) and is accessible with *BACK in the ADFS.

Examples:

```
*DIR
*DIR  $.Fred.Software
```

Re-selects the URD as the CSD and sets the CSD as '$.Fred.Software'.

*DUMP

Syntax:

```
*DUMP <filename> {<offset into file> {<start address>}}
```

*DUMP opens the specified file and displays its contents in hexadecimal and ASCII as text lines in the following format:

```
Address   : 00 01 02 03 04 05 06 07 : ASCII data
00008000  : 42 32 43 01 01 0D 9C 4D : B C....M
```

Each line shows the address, hexadecimal value and ASCII value of each byte, with unprintable ASCII codes represented by full stops. The width of the displayed lines varies according to the display mode selected: it is 16 bytes wide in 80-column modes.

The address field usually shows how far through the file each byte appears, starting at the file's load address for code files or at zero for time-stamped files.

The optional 'offset into file' parameter allows dumping to begin part way through the file – useful for skipping past the beginning of files.

The optional 'start address' parameter enables you to substitute the load address of the file for a different address, making the displayed address of each byte be the start address plus the offset within the file.

*ENUMDIR

Syntax:

```
*ENUMDIR <directory> <file> {<search pattern>}
```

The *ENUMDIR command lists the objects found in the specified directory into the named file as a series of text lines delimited by line feeds (ASCII 10). It is like a file-orientated version of *CAT. The optional 'search pattern' may be used to restrict the objects listed out by means of a wildcard search.

Example:

```
*ENUMDIR & CatFile
```

Lists the contents of the CSD in the file 'CatFile'.

*EX

Syntax:

```
*EX {<directory>}
```

The *EX command 'examines' the contents of the specified directory (or the CSD if none is specified) and displays a line of information about each in the following format:

Date-stamped files:

Filename
Attributes
File type
Time
Date
Length
Disc position

Unstamped files:

Filename
Attributes
Load addr
Execute addr
Length
Disc position

*EXEC

Syntax:

```
*EXEC {<filename>}
```

*EXEC allows you to tell the OS to take keyboard input from the named file instead of the keyboard. The file is opened for input and characters are read from it until the end of the file is reached, when it is closed automatically and input restored to the keyboard.

The *EXEC function may be terminated early by the inclusion of '*EXEC' without a filename. Alternatively, including another *EXEC <filename> command will close the current file, open the new one, and start taking input from the new file instead.

Example:

```
*EXEC !Boot
```

Switches character input to the file "!Boot".

*INFO

Syntax:

```
*INFO <object name>
```

The *INFO command behaves in much the same way as the *EX command but deals with an object name (optionally including wildcards) rather than a directory name. Thus '*INFO *' is equivalent to '*EX'.

Example:

```
*INFO SomeFile
```

Displays detailed information about 'Somefile'.

*LCAT

Syntax:

```
*LCAT {<subdirectory>}
```

*LCAT displays a catalogue of the currently selected library, or one of its subdirectories if a parameter is supplied.

*LEX

Syntax:

```
*LEX {<subdirectory>}
```

*LEX displays file information for the currently selected library, or one of its subdirectories, in the same way as *EX. On its own *LEX is equivalent to '*EX %'.

*LIB

Syntax:

```
*LIB {<directory>}
```

*LIB sets the currently selected library (CSL) to the named directory, so that it will be searched when a file to be *RUN is not found in the CSD. The CSL is represented by the symbol '%' and it is necessary to set the OS variable Run$Path to include '%' so that the library is searched.

*LIST

Syntax:

```
*LIST <filename>
```

The *LIST command displays the contents of the specified file in the format specified by *CONFIGURE DumpFormat. Each line is displayed with a line number.

*LOAD

Syntax:

```
*LOAD <filename> {<load address>}
```

The *LOAD command loads the specified file into memory. If no load address is provided the file's existing load address is used; otherwise the file is loaded at the address supplied. If the file is Date stamped, then the action of load changes. The OS looks for a system variable of the type Alias$@LoadType_ttt, where 'ttt' is the type of the file being loaded. If the variable exists then the OS 'executes' the string assigned to the variable by sending it to OS_CLI.

For example, by default, the variable 'Alias$@LoadType_FFB BASIC – Load %0' is set up by the OS. All BASIC programs are saved in files with are stamped with the file type FFB. Thus if we *LOAD a BASIC program, the OS actually performs *BASIC –load <filename>. This automatically starts up BASIC with the specified BASIC program resident in memory.

Several known file types are aliased in this way so that appropriate action is taken if a date stamped file of the corresponding type is *LOADed. A similar scheme is used to deal with attempts to *RUN a date stamped file.

You can ues *SHOW to display the current file type settings.

*OPT

Syntax:

```
*OPT <option number> {<value>}
```

The *OPT command is used to set various filing system options for the current filing system. These are as follows:

*OPT 0 Resets the *OPT options to their defaults

*OPT 1,<x> Affects whether *INFO-style file information is displayed when files are loaded or saved. The value of <x> is interpreted as follows:

0 No information displayed.
1 The filename is displayed.
2 The filename and its load address, execution address and length are displayed.
3 Either the above, or file type and date-stamp information are displayed, according to the file type.

*OPT 4,<x> Sets the auto-boot option as follows:

0 No auto-boot
1 *LOAD the boot file
2 *RUN the boot file
3 *EXEC the boot file

The boot file is either !Boot (for the ADFS) or !ArmBoot (for the NFS).

*PRINT

Syntax:

```
*PRINT <filename>
```

The *PRINT command opens the named file for input and sends the contents of the file to the VDU drivers. This means that ASCII control codes will have their VDU effect, rather than being displayed as split-bar '|' sequences.

*REMOVE

Syntax:

```
*REMOVE <object name>
```

This command operates as for *DELETE except that no error is generated if the specified object does not exist. Note that wildcards are not permitted – you should use *WIPE instead.

Example:

```
*REMOVE NoSuchFile
```

Appears to delete the file 'NoSuchFile' without error.

*RENAME

Syntax:

```
*RENAME <current object name> <new object name>
```

The *RENAME command changes the pathname by which the specified object is known. It can, therefore, not only change the name of an object, but also 'move' it within the directory hierarchy. The object must already exist and its new name must not exist in the target directory.

Examples:

```
*RENAME Joules Carol
```

Changes the name of file 'Joules' to 'Carol'.

```
*RENAME &.Physics.Nicko $.Exams.Power
```

Moves the file 'Nicko' from the subdirectory 'Physics' of the CSD into $.Exams and renames it 'Power'.

*RUN

Syntax:

```
*RUN <filename> {<parameters>}
```

*RUN both loads and then executes the specified file, using the load and execute addresses associated with the file. Parameters may optionally be added which are accessible to the program when it begins execution.

Example:

```
*RUN  4YourLife
```

Loads and executes the file '4YourLife'.

*SAVE

Syntax:

```
*SAVE <filename><start addr><end addr>{<exec addr>{<reload addr>}}
```

or alternatively:

```
*SAVE <filename><start addr>+<length>{<exec addr>{<reload addr>}}
```

*SAVE copies the specified area of memory to a named file. The memory area and the file's associated load and execute addresses may be specified in one of the two forms shown above. In the first case the start address and the address of the byte *after* the last byte to be saved are supplied; in the second case the start address and the length in bytes are supplied.

A reload address may optionally be supplied, causing the file thereafter to be loaded at a different address from where it is saved.

Where execute and reload addresses are not supplied, defaults of the start address for saving are assumed.

Example:

```
*SAVE TheWorld 0 +40000
```

Saves 256k of memory into file 'TheWorld'.

*SETTYPE

Syntax:

```
*SETTYPE <filename> <type>
```

*SETTYPE allows the file type set by the OS for the file to be changed. The 12-bit 'type' is applied to the named file, normally entered as three hexadecimal digits.

Example:

```
*SETTYPE Alphabet ABC
```

Sets the file type of file 'Alphabet' to &ABC.

*SHUT

Syntax:

```
*SHUT
```

This command has the same effect as *CLOSE (ie, it closes all open files) but it affects all filing systems rather than just the currently selected one.

*SHUTDOWN

Syntax:

```
*SHUTDOWN
```

This command has an even broader effect than *SHUT. It performs all the functions of *SHUT and also logs the user off any network file servers in use and dismounts any discs, both floppy and hard, so leaving the computer in a 'disconnected' state. The use of *SHUTDOWN at the end of working sessions is highly recommended.

*SPOOL

Syntax:

```
*SPOOL {<filename>}
```

The *SPOOL command opens the named file (the 'spool file') and sends all subsequent VDU output to it. In many ways *SPOOL is the inverse operation of *EXEC. If the file already exists then its contents are overwritten.

*SPOOLON

Syntax:

```
*SPOOLON {<filename>}
```

*SPOOLON has the same effect as *SPOOL, except that VDU output is appended to the file if it exists, rather than overwriting it. If the file does not already exist, *SPOOLON is exactly equivalent to *SPOOL.

Either *SPOOL or *SPOOLON without a parameter will close the spool file.

*STAMP

Syntax:

```
*STAMP <filename>
```

*STAMP overwrites the old date and time-stamp for the specified file with the current date and time. The file type is set to &FFD if the file was not already stamped.

Example:

```
*STAMP PennyBlack
```

Stamps the file 'PennyBlack' with the current date and time.

*TYPE

Syntax:

```
*TYPE <filename>
```

The *TYPE command has a similar effect to the *LIST command, ie, it displays the contents of the file in the way defined by the configuration 'DumpFormat', but does not precede each line with a line number.

Example:

```
*TYPE Pitman
```

Displays the contents of file 'Pitman' on the screen.

*UP

Syntax:

```
*UP {<how_far>}
```

The *UP command moves a specified number of levels 'up' through the directory hierarchy of the currently selected filing system – equivalent to *DIR followed by a sequence of caret symbols '^'.

Example:

```
*UP 2
```

Performs a *DIR ^.^

*WIPE

Syntax:

```
*WIPE <object spec.> {<options>}
```

The *WIPE command deletes the specified object(s) with wildcards being permitted. This allows several files or the contents of directories to be deleted in one go. A number of options may also be included:

C Confirm. Prompts you for confirmation before each deletion.

F Force. Deletes objects even if they are locked.

R Recurse. Deletes subdirectories and their contents as well.

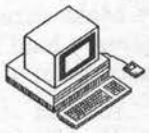
V Verbose. Displays file information prior to deletion.

Operating System variable Wipe$Options contains the default settings for these options – they may be overridden when issuing *WIPE by specifying the desired attribute set after the command. Again use a tilde '~' to de-select a particular feature, eg:

```
*WIPE my_file ~C
```

No confirmation required.

9 : Filing System SWIs



In addition to the command line interface the FileSwitch module also provides a range of SWIs which allow the same effects to be achieved efficiently from within programs. Just as many of the '*' commands bear a strong resemblance to BBC MOS commands, so the Operating System filing system SWIs resemble BBC MOS calls.

The BBC MOS defines a general interface to filing systems through routines known as OSFILE, OSFIND, OSARGS, OSGBPB, OSBGET and OSBPUT. These are provided under the OS using SWIs with similar names, viz: OS_File, OS_Find, OS_Args, OS_GBPB, OS_BGet and OS_BPut. For most of these routines the parameters each require are taken in the same order as for the BBC MOS. The main difference is that under the OS, parameters are passed in the ARM's registers instead of in a table of bytes in memory as is the case with the BBC MOS. The general rule for converting software from the BBC MOS form to work on this OS is to take the parameters in the order they appear in the BBC MOS table and place them in successive ARM registers.

These SWIs are outlined below although only the situations where they behave significantly differently from their original BBC MOS form are highlighted.

OS_File (SWI &08): Operate on Entire Files

OS_File SWI deals with loading, saving and modifying the attributes of whole files. Register R0 contains a 'function code' indicating what action is to be taken. Other, successive registers contain the required parameters:

R0=&00 Save memory to file

R1=pointer to filename string
R2=load address of file
R3=execution address of file
R4=start address of data in memory
R5=end address of data in memory

As an example, the program given below will save an entire MODE 12 screen in a file called SCREEN. Because it is accessing screen memory directly it is

much faster than the corresponding ScreenSave command provided by the OS. Make sure you have 80k left on your disc to hold the screen! A complementary load screen program listing is given later in this chapter under the description of the the OS_File SWI.

```
 10 REM >List9/1
 20 REM Example of OS_File to save
 30 REM screen memory
 40 REM (c) Mike Ginns 1988
 50 REM Archimedes OS: A Dabhand Guide
 60 REM Dabs Press
 70 :
 80 DIM ScreenSave 1024
 90 FOR pass = 0 TO 3 STEP 3
100 P%= ScreenSave
110 [
120 OPT pass
130
140
150 ADR  R0,Pblock          \ Read Start address of screen
160 ADD  R1,R0,#8
170 SWI  "OS_ReadVduVariables"
180 LDR  R4,Pblock+8        \ Start address for SAVE now in R4
190
200 ADD  R5,R4,#1024*80     \ R5 (end address) = R4 + 80 Kbytes
210
220 MOV  R0,#0              \ Select Save OS_File option
230 ADR  R1,filename        \ Make R1 point to file name
240 MOV  R2,#0              \ zero re-load address
250 MOV  R3,#0              \ zero execution address
260
270 SWI  "OS_File"          \ Call OS_File to perform the SAVE
280
290 MOV  PC,R14             \ Return to BASIC
300
310 .filename
320 EQUS  "Screen"
330 EQUB  0
340
341 ALIGN
350 .Pblock
360 EQUD  149               \ OS Var. number for start of screen
370 EQUD  -1
380
390 ]
400 NEXT
410
420 MODE 12
430
440 FOR T%= 0 TO 300
450 GCOL 0,RND(7)
460 CIRCLE FILL RND(1280),RND(1024),RND(150)
```

```
470  NEXT
480
490  PRINT"Saving screen now"
500  CALL ScreenSave
510  PRINT"Screen saved"
```

Listing 9.1. Save screen using OS_File SWI.

R0=&01 Write Catalogue Information

This call makes it possible to change the catalogue information about a specified file. The new catalogue information is held in the same register as in the previous command. However, this time R5 contains a series of flags which specify the access type to the file. The format of these attribute flags is as follows:

bit 0	File has read access for you
bit 1	File has write access for you
bit 2	Not used
bit 3	File is locked
bit 4	File has read access for others
bit 5	File has write access for others
bit 6	Not used
bit 7	Not used

The concept of 'you' and 'others' is only relevant in the NFS system. In ADFS bit 0 and 4 should be the same, as should bits 1 and 5.

R0=&02 Write Load Address Only

Address supplied in R2.

R0=&03 Write Execution Address Only

Address supplied in R3.

R0=&04 Write Attributes Only

Attribute flags in R5 as shown above.

R0=&05 Read Catalogue Information

R1=pointer to filename string

The information is returned in R0-R5:

R0=object type (0=not found, 1=file, 2=directory)
R1=(the same) pointer to filename string
R2=load address
R3=execution address
R4=length
R5=access attributes (bottom byte)

As an example the following program (listing 9.2.) will prompt for a filename, read its catalogue details and then display them.

```
  10  REM >List9/2
  20  REM Example of OS_File to get
  30  REM file information.
  40  REM (c) Mike Ginns 1988
  50  REM Archimedes OS: A Dabhand Guide
  60  REM Dabs Press
  70
  72 REPEAT
  80 INPUT filename$
  90 SYS "OS_File",5,filename$ TO type,, load, execution, length,
attributes
 100
 110  IF type =0 THEN PRINT " Could not find the specified object"
 120  IF type =2 THEN PRINT " Object is a Directory"
 130  IF type =1 THEN
 140  PRINT " Object is a File"
 150  PRINT " Load address is  : "~load
 160  PRINT " Execution address is : "~execution
 170  PRINT " Length of file is : "length
 180  IF attributes AND 1 PRINT " File has read access"
 190  IF attributes AND 2 PRINT " File has write access"
 200  IF attributes AND 8 PRINT " File is locked"
 210  ENDIF
 220
 230  UNTIL0
```

Listing 9.2. Use of OS_File to read catalogue information.

R0=&06 Delete an Object

Returns catalogue information as above, after deleting the specified object.

R0=&07 Create an Empty File

As for R0=0, but the start and end addresses in R4, R5 are used only to determine the size – no data is written.

R0=&08 Create a Directory

This new call has the same effect as *CDIR: it takes a zero-terminated name string pointed to by R1 and the minimum number of entries the directory should contain in R4 (zero will provide the default).

Note the minimum number of directory entries is not relevent to ADFS and is ignored. It is, however, of use with the NFS system.

R1=pointer to directory name string
R4=minimum number of entries (zero for default number)

As an example the following code fragment will create a sub-directory called "Richy" in the root directory ($).

```
ADR R1,filename
MOV R4,#0
MOV R0,#8
SWI "OS_File"
...
.filename
EQUS    "$.Richy"
EQUB    0
```

R0=&09 Write Date/Time Stamp

This call acts exactly the same as *STAMP; it takes a filename string and applies the current date/time to it. The file has a file type of &FFD after the call has been used.

R1=pointer to filename string

R0=&0A Save Memory to Date/Time Stamped File

This is just the same as the R0=0 case, but since the load and execute addresses are not required, only the file type is needed (in R2).

R1=pointer to filename string
R2=file type (bottom 12 bits)
R4=start address of data in memory
R5=end address of data in memory

R0=&0B Create a Date/Time Stamped File

This is the same as the R0=7 case, but again the parameter in R2 is the file type since the load and execute addresses are not required.

R0=&0C Load File with Path String

Files are usually searched for using the path sequence set by the Operating System variable File$Path. This call allows File$Path to be overridden by providing a path string pointed to by R4. The other parameters are as for R0=&FF (see below).

R1-R3 As for R0=&FF
R4=pointer to path string

R0=&0D Read Catalogue Information with Path String

This is the catalogue form of the above, so it is equivalent to R0=5 except that R4 holds a pointer to the chosen path string which overrides File$Path. The results are returned just as for R0=5.

R1=pointer to filename string
R4=pointer to path string

R0=&0E Load File with Path Variable

This is the same as R0=&0C except that it requires the path string to be held in an OS variable, so R4 points to this variable instead of an immediate string. R1-R3 are as for R0=&FF (see below).

R0=&0F Read Catalogue Information with Path Variable

This is the same as R0=&0D except that the path string must be in an OS variable pointed to by R4.

R0=&10 Load File Using No Path

The minimal version: the filename is taken as supplied, with no path being used to prefix it. Otherwise, this is the same as R0=&0C, and its parameters are as for R0=&FF (see below).

R0=&11 Read Catalogue Information Using No Path

The same as R0=&0D except that no path string prefixes the filename.

R0=&12 Set File Type Only

In much the same way as for R0=2 to 4, this call allows just the file type to be set.

R1=pointer to filename
R2=file type (bottom 12 bits)

R0=&FF Load File Into Memory

This call is the general purpose file loading SWI. It reads a file's catalogue information and then loads it at one of two addresses: either the one supplied by the file's information, or an overriding address in R2. Which of the two addresses is used is determined by the bottom byte of R3: if zero, the override in R2 is used, otherwise the file's own load address is used.

R1=pointer to filename string
R2=load address (if R3 lsb=0)
R3=file/override flag (see above)

On return the registers are filled as for R0=&05, eg:

R0=1 (object type is file)
R1=(the same) pointer to filename string
R2=load address
R3=execute address
R4=length
R5=access attributes

The program listed below (listing 9.3) will load a data file directly back into a screen memory. When RUN, it tries to reload the MODE 15 screen saved by the program given in OS_File (R0 = 0), ie, listing 9.1.

Try using the OS ScreenLoad and ScreenSave commands to perform the same operation and notice the speed difference!

```
 10  REM >List9/3
 20  REM Example of OS_File to load
 30  REM screen memory
 40  REM (c) Mike Ginns 1988
 50  REM Archimedes OS: A Dabhand Guide
 60  REM Dabs Press
 70
 80  DIM ScreenLoad 1024
 90  FOR pass = 0 TO 3 STEP 3
100  P%= ScreenLoad
110  [
120  OPT pass
130
```

```
140  ADR R0,Pblock          \ Get address of start of screen
150  ADD R1,R0,#8
160  SWI "OS_ReadVduVariables"
170  LDR R2,Pblock+8        \ Load address for file now in R2
190
200  MOV R0,#&FF            \ Select OS_File LOAD option
210  ADR R1,filename        \ File name to load
220  MOV R3,#0              \ Over-ride files own load address
230  SWI "OS_File"          \ Call OS_File to perform LOAD
240
250  MOV PC,R14             \ Back to BASIC
260
270  .filename
280  EQUS "Screen"
290  EQUB 0
300
301  ALIGN
310  .Pblock
320  EQUD 149               \ OS var. number for start of screen
330  EQUD -1
340
350  ]
360  NEXT
370
380  MODE 12
390
400  PRINT TAB(10,10) "Loading screen now"
410  CALL ScreenLoad
```

Listing 9.3. Load a block of screen memory.

OS_Find (SWI &0D) Open/Close File for Byte Access

OS_Find SWI (and its related BBC MOS routine OSFIND) allow the programmer to inform the filing system that a file needs to be made available for byte or block access. This is known as 'opening' a file. The converse 'closing' operation must be performed when the file operation is completed. Opening a file causes the filing system to translate a filename string into an integer 'handle' – a number representing the file which is used extensively by other file operations. To close a file, this handle must be supplied to uniquely identify the file. OS_Find's only purposes are the opening and closing of files.

Currently, the file manager allows 24 files to be open at any one time. A given file can be opened for read access up to 24 different times, allowing multiple read access to the same file. However, a file can only be opened once for write access. If a file has been opened for write access then it

cannot be opened for read access. Similarly if a file is already open for read access then it cannot be opened for write access.

In general for OS_Find, R0 again defines the type of action and R1 points to the filename or contains the handle (depending on the operation). R2 points to a path string, where the option to use it is chosen.

R0=&00 Close a File

R1=0	All open files on the current filing system are closed
R1>0	R1 is taken as a handle of a file which is updated from buffers in RAM and then closed

R0=&4x Open File for Input

R1 is a pointer to the filename, and the resultant handle is placed in R0 on exit. The file must already exist, otherwise a handle of zero is returned.

R0=&8x Create and Open File For Output

Again, R1 is a pointer to the filename and the handle is returned in R0. If the named file exists, it is opened for output and the file pointer and length are reset to zero. If the file doesn't exist, it is created and opened.

R0=&Cx Open File for Input/Output

This is the same as R0=&4x except that it allows output to the file as well as input. A handle of zero is returned if the file does not already exist.

See the discussion of paths below.

File Path Considerations in OS_Find

Because of the OS's new features with respect to file paths, a modification to these functions to cater for paths is provided. Where a path prefix is desired, a pointer to it is placed in R2 and the appropriate value from the table below added to the base R0 function number (see above):

Value	Meaning
0	Use File$Path to prefix the specified filename
1	Use the string pointed to by R2 as a path
2	Use OS string variable pointed to by R2 as a path
3	Use no path at all

Error Handling Extension

Two kinds of error handling extension have been added. The first allows an error to be raised if an attempt is made to open a directory, the norm being to permit it but disallow any operations on that handle.

The second allows an error to be raised if the specified file does not already exist, the norm being simply to return a handle of zero.

Where a path prefix or error handling extension is desired, values from the table below should be added to the base R0 function number (see above) as appropriate:

Value	Meaning
4	Cause an error if a directory is opened
8	Cause an error if the file doesn't already exist

OS_GBPB (SWI &0C) Get/Put Multiple Bytes from/to an Opened File

This SWI is the same as its BBC MOS counterpart OSGBPB, but with a number of extensions.

The first four calls provide for reading from or writing to an open file using either the file's current pointer or a new pointer which is supplied. The general form is:

R0=function code
R1=file handle
R2=memory address (for reading from or writing to)
R3=number of bytes to be transferred
R4=new pointer (where relevant)

Both R2 and R4 are updated during the call to reflect the final memory address and pointer value. For read operations, R3 returns the number of bytes *not* transferred (usually zero) and sets the Carry flag C if any bytes were not transferred.

R0=&01 Write Bytes to File at New Pointer Position

Requires valid R4.

R0=&02 Write Bytes to File at Current Pointer Position

R4 is ignored.

R0=&03 Read Bytes from File at New Pointer Position

Requires valid R4.

R0=&04 Read Bytes from File at Current Pointer Position

R4 is ignored.

The remaining OS_GBPB function codes perform miscellaneous filing system functions. Most are the same as their BBC MOS counterparts.

R0=&05 Read Title and *OPT 4 Boot Setting

This call should be made with R2 pointing to a block of memory where the results are returned as follows:

The length of the name string
The name string itself
The boot option (one byte)

R0=&06 Read CSD Name and Privilege Byte

R2 should point to a block of memory on entry. The information is returned at this point in the following format:

A zero byte
The length of the CSD name
The CSD name itself
The privilege byte

The privilege byte is used by Econet to indicate a status of 'owner' (byte=&00) or 'public' (byte=&FF). For ADFS, this byte is always zero.

R0=&07 Read CSL Name and Privilege Byte

This call is the same as R0=6 but returns details of the currently selected library.

R0=&08 Read Entries from CSD

This call returns a block of directory information comprising the names of successive objects in the CSD. Data is returned in memory at the address supplied in R2. The number of entries to be read is supplied in R3, and R4 should contain the first object number. If R4 contains zero, the first name to be returned will be the first alphabetically. The resulting data in memory is of the form:

Length of name
Name (null-terminated)
...repeated for the number of objects specified

If not all the names could be supplied, the Carry flag is set and R3 is left containing the number of those that are outstanding. Otherwise R3 contains zero and the Carry flag is clear.

R0=&09 Read Entries from Specified Directory

This call is an extended form of the previous one. As well as the parameters shown above, it also allows the directory you wish to read (by supplying a pathname pointed to by R1) to be specified. The size of the memory buffer is held in R5 (so that it doesn't overflow) and a wildcard string is pointed to by R6 (to select which entries are to be returned). Clearly this is a lot more versatile than the R0=&08 form.

Names that match the wildcard are returned in the buffer as a series of null-terminated strings, their number being returned in R3. R4 is updated to allow more entries to be read. If there are no more enteries it contains –1.

R0=&0A Read Entries and Information from Specified Directory

This call effectively performs the SWI version of *EX. Its parameters are the same as above, but it returns a block of information for each entry rather than just the name. This block is word aligned and formatted as follows:

Offset	Information
&00	Load address
&04	Execute address
&08	Length
&0C	Access attributes
&10	Object type
&14	Object name (null-terminated)

Listing 9.4 at the end of this section illustrates how this OS_GBPB call is used. The command:

```
SYS "OS_GBPB",10,dir$,data%,1,next%,63,"*" TO ,,,number%,next
```

is used to extract the details of the next directory entry. The entry parameters (from left to right) are:

OS_GBPB code
directory name
place to put file information
number of files to examine
first entry to examine
size of data block (data%)
wildcard string

the last item is that which filenames are matched against. In this case "*" is used as the wildcard string, so all files will match. After the call, the variable number% contains 0 if no more entries could be found or 1 if a valid entry existed. The variable next% is updated so that it points to the next group of entries (in this case, just the next entry).

The procedure PROCexaminedir is recursive, so that details of subdirectories are also displayed. This recursion is implemented by the command:

```
PROCexaminedir(dir$+"."+FNgetname(data%+&14),level%+1)
```

In other words, the procedure calls itself with the name of one of its subdirectories. The call OS_FSControl is used to convert the 'type' of a file into a string describing the type.

OS_BGet (SWI &0A)
Get Byte at Pointer from File

OS_BGet SWI is exactly the same as the BBC MOS call OSBGET – reading the next sequential byte from an open file whose handle is specified and then incrementing the file pointer by one. On entry, R1 should contain the file handle and on return, R0 will contain the byte read. The Carry flag 'C' is clear if the byte was read correctly and set if a problem arose (most likely through an end-of-file condition or an invalid handle).

As an example of OS_BGet the following program (listing 9.4) reads every byte in a file and counts the number of spaces. This gives, albeit approximately, indication of the number of words in the file (assuming each is separated by only one space!).

```
  10 REM >List9/4
  20 REM Example of OS_BGet to count
  30 REM spaces/words in a file
  40 REM (c) Mike Ginns 1988
  50 REM Archimedes OS: A Dabhand Guide
  60 REM Dabs Press
  70
  80
  90 DIM WordCount 1024
 100 FOR pass = 0 TO 3 STEP 3
 110 P%= WordCount
 120 [
 130 OPT pass
 140
 150 \ On entry R0 points to name of file
 160
 170 MOV R10,#0          \ total of number of spaces
 180
 190 MOV R1,R0           \ Points to file name
 200 MOV R0,#64          \ Open for read access only
 210 MOV R2,#0
 220 SWI "OS_Find"       \ Returns file handle in R0
 230
 240 MOV R8,R0
 250 .count_loop
 260 MOV R1,R8
 270 SWI "OS_BGet"       \ Get byte from file
 280 BCS quit_loop       \ If EOF quit loop
 290 CMP R0,#32          \ Is it a space ?
 300 ADDEQ R10,R10,#1    \ If so, increment the count
 310 B count_loop        \ If not end of file then loop
 320
 330 .quit_loop
 340 MOV R1,R8
 350 MOV R0,#0
 360 SWI "OS_Find"       \ Close file
 370
 380 MOV R0,R10          \ Make word count available to USR
 390
 400 MOVS PC,R14
 410
 420 ]
 430 NEXT
 440
 450 MODE 0
 460 DIM filename 16
 470 INPUT "Please enter the filename to be investigated : "
$filename
 480
 490 PRINT "Counting words now"
 500 A% = filename
```

```
510  words = USR( WordCount )
520  :
530  PRINT "Number of words counted is : " words
```

Listing 9.4. Using OS_Bget to count spaces and words.

OS_BPut (SWI &0B) Put Byte at Pointer to File

This call performs the reverse of OS_BGet and is equivalent to the BBC MOS call OSBPUT. It writes the byte passed in R0 to the file whose handle is supplied in R1 at the current file pointer, and then increments the file pointer by one to allow further OS_BPut calls to work correctly.

OS_Args (SWI &09) Read/Write Open File Information

This is another of the general-purpose SWIs and has an equivalent under the BBC MOS, although the meanings of most of the operations differ and, therefore, are all documented here.

The general format of the call is that on entry, R0 contains a function code, R1 contains a file handle and R2 contains data to be written (for write operations). Usually all registers are preserved, except where information is being read, in which case it is returned in R2.

R0=&00 Read Sequential File Pointer/Filing System Number

To make life difficult this function code has two meanings, making it a special case. When entered with R1=0 (instead of a valid file handle) it returns the number of the current filing system in R0 (using the BBC MOS numbering strategy).

If R1 holds a valid file handle (ie, a non-zero value) this call returns the current value of the sequential pointer of that file in R2.

R0=&01 Write Sequential File Pointer

The reverse of the above call, it allows the sequential file pointer to be written (set). It should be entered with a valid file handle in R1 and the new file pointer value in R2. If the new pointer value is beyond the current extent of the file then its size is increased accordingly and the new area filled with zeros.

R0=&02 Read File Extent

This call allows the current file extent (length) to be read into R2 for the file whose handle is supplied in R1.

R0=&03 Write File Extent

This call allows the extent of the file whose handle is supplied in R1 to be set to the value in R2. If this value is larger than the existing extent, the file is extended accordingly and the new area filled with zeros.

R0=&04 Read Allocated Size

This call allows the amount of space actually allocated to the file to be read – enabling you to determine how much space is left before new space need be allocated. It returns in R2 the allocated space for the file whose handle is supplied in R1.

R0=&05 Read End-of-file Status

This is one of two ways of sensing the end-of-file condition (the alternative being OS_Byte call &7F provided for BBC MOS compatibility). When supplied with a file handle in R1, this call returns a non-zero result in R2 if the file pointer is equal to the file's extent, otherwise it returns zero.

R0=&06 Write Allocated Size

This is the reverse of the call with R0=4 (above) – allowing you to advise the Operating System that at least the amount of space in R2 should be reserved for the file whose handle is supplied in R1. On return, R2 contains the amount of space actually allocated.

R0=&FF 'Ensure' File Buffers

This call, carried over from the BBC MOS, is equivalent to part of the 'closing' process for files (under OS_Find). It ensures that all filing system buffers are written out to their corresponding files, thus allowing them to be closed. On entry, R1 must contain either zero (in which case all files are 'ensured') or a valid file handle (in which case just the chosen file's buffers are 'ensured').

OS_FSControl (SWI &29)
General Filing System Control

This call provides a wide range of different actions to be performed on the selected filing system, including setting the CSD or CSL, *RUNning a file, *CATaloging a directory and so on. The most useful of these routines are described in the following section.

On entry to OS_FSControl, register R0 contains a number which specifies which of the many operations is to be carried out. The contents of other registers depend on the action being performed and are described individually in the following sections.

R0=&00 Set Current Directory

This call is used to change the currently selected directory (CSD) for the file system. It is equivalent in effect to the Operating System command *DIR <dirname>. On entry to the routine, R1 must point to a zero-terminated string which contains the name of the directory to be selected. If the directory name is null, then the current directory reverts to the root directory as default.

R0=&01 Set Library Directory

This call is similar to the above, but is used to change the currently selected library directory (CSL). It is equivalent in function to the Operating System command *LIB <dirname>. On entry to the routine, R1 must point to a terminated string which contains the name of the directory to be selected as the library. If the directory name is null, then the library directory reverts to a default, typically $.Library for ADFS (if this is present).

R0=&02 Reserved for Operating System

Do not use.

R0=&03 Reserved for Operating System

Do not use.

R0=&04 RUN file

This routine will *RUN the named file. The name of the file to be loaded and executed must be contained in a terminated string which is pointed to by

register R1. The file is searched for in the directories specified in the system variable Run$Path. By default, this is set up to be the current and then the library directory.

If the file being *RUN is date stamped then a suitable RUN alias is looked for which corresponds to the file type. See the section on *LOAD for a description of the action taken.

R0=&05 Catalogue a Directory

This routine performs an equivalent function to the command *CAT <dirname>. The name of the directory to be catalogued is contained in a terminated string which is pointed to by register R1. If this name is null, then the currently selected directory is catalogued. For example:

```
SYS "OS_FSControl",5,""
```

R0=&06 Examine Current Directory

This call prints out full catalogue information on each file in the specified directory. It is therefore equivalent to the command *EX <dirname>. The name of the directory to be examined is contained in the null terminated string pointed to by register R1.

R0=&07 Catalogue Library Directory

This performs a similar function to the call with R0=&05 except that it displays a catalogue on the currently selected library directory like *LEX. Again R1 points to a terminated directory name. If this is null then the current library directory itself is catalogued. Otherwise the name is taken to be that of a sub-directory within the library directory which is catalogued instead.

R0=&08 Examine Library Directory

This call is similar to that with R0=&06 except that information on files in the current library directory is displayed. If R1 points to a null string, then the files in the library directory itself are examined. Otherwise it is assumed to point to the name of a sub-directory within the library. The catalogue details of the files in this sub-directory will then be printed instead.

R0=&09 Examine Specified Objects

This call allows a, possibly ambiguous, file name and path to be specified. It then prints out information on any file which matches this specification. On entry R1 points to the file name/path to be used. For example, if R1 pointed to the string "A*", then information would be printed on all files beginning with an 'A' in the current directory. Similarly "$.Richy.A*" would examine every file beginning with an "A" in the sub-directory "Richy".

R0=&0A Set File System Options

This call is equivalent to the command *OPT n,m where 'n' is the option number to be set and 'm' is the value. On entry, R1 contains the option number (n) and R2 the parameter value (m). For example, if R1 contained four and R2 contained three, then the call would perform *OPT 4,3 and select a *EXEC boot option. If R1=0 then the settings of all of the file system options are reset to their default state. For example:

```
MOV R0,#&0A \ Set * OPT 4,3
MOV R1,#4
MOV R2,#3
SWI "OS_FSControl"
```

R0=&0B Set File System from Named Prefix

This call sets the currently selected file system to be that specified in the string pointed to by R1. If this string does not contain a valid file system name then no action is taken. As an example, consider the following program:

```
MOV R0,#&0B
ADR R1,string
SWI "OS_FSControl"

.string
EQUS "ADFS:fred"
EQUB 0
```

The file manager will recognise the file system name within the string and select the ADFS system.

On exit from the routine the following registers return information.

R1 Points to immediately after the file system's name if one was present in the string.

R2 Equals -1 if no file system name specification was found.

R3 Points to a special field if one was present.

Note that 'name:' is the preferred way of specifying a file system name within a command. However, the older alternative '–name–' can be used instead.

R0=&0C Add File System

Add a new file system to those recognised by the file manager. This call is only of interest to readers considering writing a completely new file system.

R0=&0D Lookup File System

This call returns information about a file system. On entry R1 identifies the file system to be investigated. R2 specifies how the file system name is terminated. On exit : R1 = file system name and R2 points to the file system control block. This call is only of interest to readers considering writing a completely new file system.

R0=&0E Select File System

This call selects a specified file system to be the current one. This call is only of interest to readers considering writing a completely new file system.

R0=&0F Boot File System

This call requests the currently selected file system to perform the boot operation. This is done when SHIFT-BREAK is pressed. The exact nature of the boot operation depends on the file system but usually involves auto-running a file.

R0=&10 File System Removal

Removes the specified file system from those recognised by the File manager. This call is only of interest to readers considering writing a completely new file system.

R0=&11 Add Secondary File System Module

Add a secondary module to the main file system module. This call is only of interest to readers considering writing a completely new file system.

R0=&12 Translate File Type Number to Name

This call allows a file type number to be converted into the corresponding file type name. On entry R2 contains the file type number. On exit, registers R2 and R3 contain eight bytes which are the ASCII representation of the file type name. As an example, if R2 = &FFB, then R2, R3 would contain the following bytes on exit:

&49534142 20202043

These bytes are the ASCII representation for the characters "BASIC " – the name of the file type.

The program below prompts for a file type number and attempts to convert it to a file type name. The name is then printed out.

```
 10  REM >List9/5
 20  REM Example of OS_FSControl to
 30  REM convert a file type number
 40  REM (c) Mike Ginns 1988
 50  REM Archimedes OS: A Dabhand Guide
 60  REM Dabs Press
 70  :
 80  DIM work 8
110  REPEAT
120  PRINT'
130  INPUT "Please Enter File type number : &" type$
140  PRINT
150  :
160  type=EVAL("&"+type$)
170  SYS "OS_FSControl",&12,,type TO ,,byte1,byte2
180  :
190  !work=byte1
200  work?4=13
210  PRINT $work;
220  :
230  !work=byte2
240  work?4 = 13
250  PRINT $work
260  UNTIL FALSE
```

Listing 9.5. Use of OS_FSControl to convert a file type number.

R0=&13 Restore Current File System

This call takes no entry parameters. It selects the current temporary file system to be the currently selected one.

R0=&14 Reserved for the Operating System

Do not use this call.

R0=&15 Return File System Handle

When dealing with files we usually identify them using a file handle provided by the file manager. This call translates this file manager's file handle into the corresponding one actually used by the selected file system. On entry R1=file manager's file handle. On exit R1 = the corresponding handle as used by the file system.

R0=&16 Shut

This call provides an equivalent function to the *SHUT command. It closes all files on the file system.

R0=&17 ShutDown

This call provides an equivalent function to the *SHUTDOWN command. It closes all files on the file system. In addition it logs off all file servers and dismounts any ADFS discs.

R0=&18 Set File Attributes from String

This call provides an equivalent function to the *ACCESS command. It allows the attributes for any named files to be set. On entry, R1 points to a string specifying the files to be affected. This may include a wild card to affect several files. R2 points to a string which contains the new attributes to be set.

R0=&19 Rename Objects

The call performs a *RENAME. On entry R1 points to the first file/path name and R2 points to the second. The call then renames the first specified file as the second file specification.

R0=&1A Copy Object

The call provides a general file copy facility. On entry the following registers contain information:

R1	Pointer to first file/path name
R2	Pointer to second file/path name

R3	Action mask
R4	Optional start date
R6	Optional start date
R7	Optional end date
R8	Optional end date

The call is the equivalent of *COPY. All files matching the first file/path specification are copied to the second file/path specification. Various options for the copy can be specified in the flags contained in R3. The flags are as follows:

Bit	Function
8	Set to select printing of information during copy
7	Set if the original file is to be deleted after the copy
6	Set if user is to be prompted to change disc as required
5	Set if copy is allowed to use application space to speed up copy
4	Set to select verbose mode during copy
3	Set if user is to be prompted to confirm each copy
2	If set, only files between the given time/date stamps are to be copied
1	If set, locked files are unlocked and overwritten by the copy
0	Set to allow recursive copying of file through sub-directories

R0=&1B Wipe Objects

This routine provides an equivalent operation to the *WIPE command. On entry the registers must be set up as follows:

R1	Pointer to file/path name to delete
R3	Action mask
R4	Optional start date
R6	Optional start date
R7	Optional end date
R8	Optional end date

The option flags in R3 are the same as those used in the COPY routine.

R0=&1C Count Objects

This routine provides an equivalent operation to the *COUNT command. On entry the registers must be set up as follows:

R1	Pointer to file/path name to count
R3	Action mask
R4	Optional start date

R6	Optional start date
R7	Optional end date
R8	Optional end date

The option flags in R3 are the same as those used in the COPY routine. On exit from the routine R2 contains the total number of bytes counted in all matching files. R3 contains the number of matching files counted.

To illustrate the many and varied filing system calls, listing 9.6 is included at the end of this section. The operation of the program is self-explanatory and demonstrates how to use the most common forms of the more useful filing system SWIs.

```
  10  REM >List9/6
  20  REM TypeTree
  30  REM by Nicholas van Someren
  40  REM Archimedes OS: A Dabhand Guide
  50  REM (c) Copyright AvS and NvS 1988
  60  REM Set aside some workspace and terminate the
  70  REM file type string with a Return.
  80  :
  90  DIM data% 63,typebuf% 10
 100  typebuf%?8=13
 110  :
 120  REM Input directory name, using CSD by default,
 130  REM and examine it.
 140  :
 150  INPUT"Which Directory ?"dir$
 160  IF dir$="" dir$="@"
 170  PROCexaminedir(dir$,0)
 180  END
 190  :
 200  REM Examine the directory dir$. The depth of the
 210  REM examination is level%. next% is the next
 220  REM catalogue entry to be read and number% is the
 230  REM number of entries that were actually read.
 240  :
 250  DEF PROCexaminedir(dir$,level%)
 260  LOCAL next%,number%
 270  next%=0
 280  :
 290  REM While there is another entry to read, read its
 300  REM name and catalogue information and store it at
 310  REM data%. If another was found (number%>0) then
 320  REM print its name and determine its type.
 330  :
 340  WHILE next%<>-1
 350  SYS "OS_GBPB",10,dir$,data%,1,next%,63,"*" TO
,,,number%,next%
 360  IF number%>0 THEN
 370  PRINTTAB(10);":"TAB(level%*3+12);FNgetname(data%+&14);
```

```
 380  VDU 13
 390  :
 400  REM If the entry was a directory examine it.
 410  :
 420  IF data%?&10=2 THEN
 430  PRINT"Directory"
 440  PROCexaminedir(dir$+"."+FNgetname(data%+&14),level%+1)
 450  :
 460  REM If it was a file, and the top 12 bits of the
 470  REM load address are &FFF, find the file type.
 480  REM Otherwise, print it as 'code'.
 490  :
 500  ELSE
 510  IF (!data% >>> 20)=&FFF THEN
 520  SYS "OS_FSControl",18,,(!data% >>> 8)AND &FFF TO
,,!typebuf%,typebuf%
 530  PRINT$typebuf%
 540  ELSE
 550  PRINT"Code"
 560  ENDIF
 570  ENDIF
 580  ENDIF
 590  ENDWHILE
 600  ENDPROC
 610  :
 620  REM Extract the file name stored at addr%.
 630  :
 640  DEF FNgetname(addr%)
 650  LOCAL b$
 660  WHILE ?addr%>31
 670  b$+=CHR$(?addr%)
 680  addr%+=1
 690  ENDWHILE
 700  =b$
```

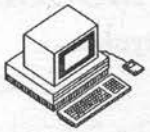
Listing 9.6. Display directory tree.

```
  10  REM >List9/7
  20  REM FileTest
  30  REM by Nicholas van Someren
  40  REM Archimedes OS: A Dabhand Guide
  50  REM (c) Copyright AvS and NvS 1988
  60  :
  70  DIM text% 1000
  80  endtext%=text%
  90  PRINT "Please enter some lines of text, pressing Return"
 100  PRINT "after each. Press Return by itself to end."
 110  INPUT LINE a$
 120  WHILE (endtext%+LEN(a$)<text%+1000) AND a$<>""
 130  $endtext%=a$
 140  endtext%+=LEN(a$)+1
 150  INPUT LINE a$
```

```
 160  ENDWHILE
 170  PRINT"Thank you - saving text..."
 180  SYS "OS_File",0,"TextFile",0,0,text%,endtext%
 190  *INFO TextFile
 200  PRINT"Change all the file information:"
 210  SYS "OS_File",1,"TextFile",&FFFFFE12,&3456789A,,8
 220  *INFO TextFile
 230  PRINT"Change the load address:"
 240  SYS "OS_File",2,"TextFile",0
 250  *INFO TextFile
 260  PRINT"Change the execution address:"
 270  SYS "OS_File",3,"TextFile",,&87654321
 280  *INFO TextFile
 290  PRINT"Change the attributes:"
 300  SYS "OS_File",4,"TextFile",,,,3
 310  *INFO TextFile
 320  PRINT"Open the file - ";
 330  SYS "OS_Find",192,"TextFile" TO filehandle%
 340  PRINT"the file handle is ";filehandle%
 350  PRINT"Read in a byte - ";
 360  SYS "OS_BGet",,filehandle% TO gotbyte%
 370  PRINT"the byte was ";gotbyte%
 380  PRINT"Read the extent of the file - ";
 390  SYS "OS_Args",2,filehandle% TO ,,fileextent%
 400  PRINT"the extent is ";fileextent%
 410  PRINT"Set the pointer to the end of the file:"
 420  SYS "OS_Args",1,filehandle%,fileextent%
 430  PRINT"Write back the byte we read on to the end:"
 440  SYS "OS_BPut",gotbyte%,filehandle%
 450  PRINT"Read the middle third of the file:"
 460  SYS "OS_GBPB",3,filehandle%,text%,fileextent% DIV
3,fileextent% DIV TO,,endtext%
 470  PRINT"Save the middle third back onto the end:"
 480  SYS "OS_GBPB",1,filehandle%,text%,endtext%-
text%,fileextent%+1
 490  PRINT"Force the file to be updated with:"
 500  SYS "OS_Args",&FF,filehandle%
 510  PRINT"Close the file and examine the result:"
 520  SYS "OS_Find",0,filehandle%
 530  *TYPE TextFile
```

Listing 9.7. Manipulating file attributes.

10 : Modules



The most fundamental way in which the Operating System can be expanded is by the use of 'Relocatable Modules' – pieces of software which add applications or services to the computer in a well-defined and structured way. The BBC MOS allowed users to install 'Sideways ROMs' in order to add to the computer – the module concept allows the same thing to be achieved but in a more elegant and consistent way.

The most crucial way in which modules differ from Sideways ROMs is simply that modules do not need to be physically installed in the computer. Instead, the OS maintains an area of memory – the Relocatable Module Area (RMA) – and allows modules to be loaded in to this area for execution, in any order you like. In this way the physical complexity of taking the computer apart and inserting ROMs is removed, but at the same time the elegance of expanding the system software is preserved. You should note that, in fact, it is possible to install modules in ROM: this is exactly how a large amount of the built-in software is provided, but this mechanism is not intended for use by 'mere mortals' such as we; only for Acorn themselves.

As we have seen throughout this book, a good deal of emphasis has been placed on BBC MOS compatibility by the authors of the OS. The software interface to modules is no exception in this respect – familiar ideas such as service entry points and service codes are still to be found. Clearly, in order to be able to improve upon the BBC MOS a number of features have been removed and many more added. Nevertheless, the role of the module writer is made a good deal simpler by perpetuating a lot of the old ideas.

If you have had experience of writing Sideways ROMs then you are very unlikely to encounter any difficulty in understanding how the module system is designed to work.

Module Related Commands

Before examining the detailed implementation of modules it is necessary to have a grasp of the commands which are provided to deal with module management. These commands are provided by the OS and take advantage of the module interface which we will be examining later on.

*MODULES

Syntax:

```
*MODULES
```

The *MODULES command displays a list of all the modules which are installed in the computer. No distinction is made between modules which are supplied in ROM and those which have been installed into RAM from a filing system. For each module present, a one-line entry appears which details the module's name (for use with other commands), the base address in memory where the module appears and the address of where the module's private workspace begins. Unfortunately, *MODULES does not trouble to tell us the amount of workspace consumed by each module, which is a pity – you have to work it out in your head instead!

*RMCLEAR

Syntax:

```
*RMCLEAR
```

This command clears the RMA of all modules which can safely be removed (not including system modules) and thus frees up their code space and workspace to maximise the amount of free memory in the RMA.

*RMKILL

Syntax:

```
*RMKILL <module name>
```

The *RMKILL command removes an individual module from the RMA in the same way as does *RMCLEAR – freeing its code space and workspace. System modules 'murdered' in this way revive themselves after a hard reset or upon receipt of the *RMREINIT command (see below).

*RMLOAD

Syntax:

```
*RMLOAD <filename> {<initialisation string>}
```

This command loads and initialises the specified file, which must be a valid piece of module code and must be of file type 'Module', ie, with file type code &FFA. After initialisation in this way the module will respond to its commands and to others such as *HELP in the usual way.

The initialisation string is optional and specific to the module being loaded – it can be used to request a particular amount of workspace for the module or for whatever other purpose the author of the module requires.

*RMREINIT

Syntax:

```
*RMREINIT <module name> {<initialisation string>}
```

The *RMREINIT command is used to resuscitate modules which have been *RMKILLed or *UNPLUGged (see below). It is effectively the same as the initialisation part of *RMLOAD, but of course the module must already be present in the machine (which is usually only the case for system modules).

*RMRUN

Syntax:

```
*RMRUN <filename> {<initialisation string>}
```

This command executes modules in the same way as *RUN executes raw machine code programs. The file must have a file type of &FFA and is loaded and executed if it is valid. *RMRUN is usually reserved for starting large applications rather than just system extensions.

*RMTIDY

Syntax:

```
*RMTIDY
```

As its name suggests, *RMTIDY compacts the RMA and maximises the free memory available. Because modules need to be warned when this is happening (they may be running) certain modules will initiate their own tidying up, so sounds may cease and files may be 'ensured' onto the appropriate storage medium, for example.

*UNPLUG

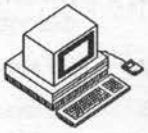
Syntax:

```
*UNPLUG {<module name>}
```

*UNPLUG allows a system module (in ROM) to be excluded from the initialisation process, rendering it completely inoperable until *RMREINITed again. Because this command alters configuration memory, the module will not reappear even after switching the power off and on. It is therefore important to remember the module's name, and for this reason entering *UNPLUG on its own will give a list of the modules which are unplugged.

There is a major SWI associated with module management that is known, logically enough, as OS_Module. OS_Module is detailed because it makes extensive reference to concepts not yet discussed. You might like to skip forwards to it if you have any doubts during what follows.

11 : Writing Modules



Writing a module is very similar to writing a Sideways ROM for the BBC MOS. However, because of the relocatable nature of modules, a number of special considerations need to be taken into account. Furthermore, several standards must be obeyed if the module is to be dealt with correctly by the OS and is therefore to run successfully.

The rest of this chapter describes how modules must respond to these various standards.

Workspace Memory

To allow a module to maintain its own status information and indeed to do its job in the broadest sense it must have access to an area of memory it can call its own. This workspace must be claimed from the RMA by calling the module manager using the SWI OS_Module with the appropriate reason code. Whenever a module is entered by the OS, register R12 is set to point to one word of memory which has been set aside as minimal workspace for that module. The OS adopts a standard that this word is used as a pointer to the actual workspace of the module (which, after all, will probably need to be larger than one word). Such workspace may therefore be addressed by using an instruction like:

```
LDR R12,[R12]
```

to get the real start address of the workspace into R12.

The use of this standard has several advantages. When *RMTIDY is issued the OS can adjust the pointer for each module that has been moved as appropriate. Also, when a module does not provide any code for shutting itself down (known as 'finalisation code') the OS can take the default action of de-allocating the workspace pointed to by the module's private word.

Module Errors

Modules also need to conform to the the OS standard for raising errors so that standard error handlers can deal with them. A module should deal with an error by following these four steps:

1. An error block (including a valid error number and a text string describing the error) should be made ready.
2. R0 should be loaded with the start address of the error block.
3. As many registers as necessary should be restored by having their previous contents pulled back from the system stack. This assumes that the original contents were preserved on the stack when the module was entered. The system stack pointer is held in R13.
4. The Overflow flag V should be set before returning.

It is important to use an error number which has been allocated specifically to the application in order that 'upstream' error handlers (which the module may not be aware of) can deal with it correctly.

The Format of Module Code

Each module is prefixed by a set of offsets into the module which the OS uses to despatch functions to each module in turn. These equate to the header information at the start of Sideways ROMs and, in fact, the services which they provide are very similar. It is crucial to remember that these are relative offsets, not absolute addresses, in order that the module be properly relocatable.

The module header always occupies the first eleven words of the module's memory space and it is divided up as follows:

Offset	Offset to
&00	Start-up code
&04	Initialisation code
&08	Finalisation code
&0C	Service call handling code
&10	Module title string (ASCII text)
&14	Module *HELP string (ASCII text)
&18	Help and command decoding table
&1C	SWI chunk base number for this module (optional)
&20	SWI handling code (optional)
&24	SWI decoding table (optional)
&28	SWI decoding code (optional)

All eleven fields must be present, though they may contain zero which the OS takes to mean that the appropriate function is not supported. The exception is the title message which *must* point to a text string terminated by zero. The fields which deal with SWIs are optional since not every

module will provide SWIs of its own. Each of the fields is discussed in detail below.

To illustrate the rather complex process of developing a piece of module software, a lengthy example – listing 11.1 – is included at the end of this section. The example is a printer buffer which, when installed, will use a specified amount of memory to store characters sent to the printer to allow you to carry on with something else while printing something out.

The listing is extensively annotated, and should you be interested in writing module software, you are advised to study the example until convinced of the concepts and rules involved.

Module Start-up Code

This is an offset to the code which will be executed if the module is started as an application, normally as a result of a *RMRUN command. Because modules do not have to provide an application it is valid for this offset to contain zero, in which case the OS will not attempt to start it up.

On entry through this offset, the CPU will be in User Mode and have interrupts enabled. Registers will not contain any useful information except for R12 which, as ever, points to the word of private workspace.

The start-up entry is used whenever OS_Module obeys a 'run' or 'enter' reason code.

Module Initialisation Code

The initialisation code is guaranteed to be called through this offset before any other part of the module is called. This obliges it to ensure that all other pointers, variables and so forth have been correctly established. It is valid for a module not to have any initialisation code, in which case zero should be placed at this offset.

Commonly, a module will use this opportunity to claim workspace, set up R12 to point to the workspace, attach to vectors, announce its presence and so on.

On entry, the CPU will be in Supervisor Mode (because it has just left the Operating System) with R13 pointing to the supervisor stack and R14 will contain a return address in the usual way. If the location pointed to by R12 is not equal to zero then this may be assumed to be a re-initialisation call (ie, not the first time around) which may have to be treated differently.

Finally, R10 points to the remainder of the command line so that user options may be decoded.

Returning from this call should be achieved by using:

```
MOV PC,R14
```

with the processor state, interrupt state and registers R7-R11 and R13 preserved. The Overflow flag 'V' should be used to specify whether an error has occurred, with R0 pointing to the error block and 'V' set if this is the case.

This offset is called whenever OS_Module obeys a 'run', 'load', 'reinit' or 'tidy' reason code.

Module Finalisation Code

This is the reverse of the initialisation entry point – it is the 'last call' before the module will be expected to expire. Usually this circumstance is reserved for situations where the module is being explicitly killed as a result of *RMKILL or because the RMA is being tidied up as a result of *RMTIDY. It is possible for a module to 'refuse to die' by causing an error in the usual way; otherwise, an exit should be taken using the link register R14 as normal. In otherwords, if the module is running, it may not be possible for it to safely shutdown. This is an error.

On entry, R10 contains a flag which indicates whether this finalisation call is because of 'true death' (from *RMKILL) or 'suspended animation' from *RMTIDY or similar). R10 contains one in the former case and zero in the latter. R12 points to the private word and R13 to the supervisor stack.

It is legitimate for a module to have no finalisation entry, in which case the OS will de-allocate its workspace on the assumption that R12 is a valid pointer to it.

This offset is called whenever OS_Module obeys a 'reinit', 'delete', 'tidy' or 'clear' reason code.

Service Call Handling Code

The service call entry point is closely related to the same entry point for BBC MOS Sideways ROMs and, in fact, many of the 'service codes' used to initiate functions are the same or similar. Service calls are initiated as a result of calls to OS_ServiceCall or the BBC MOS compatible OS_Byte &8F (whose use should be minimised). This entry point is provided for compatibility only – the OS uses the command line and SWI interfaces for most functions.

On entry to the service call handling code R1 contains the reason code of the desired service, R12 points to the private word, R13 points to a full descending stack and R14 contains a return address.

The module must decode R1 and execute software to deal with it appropriately. It is important to remember that, for much of the time, every module installed in the computer will be being 'offered' the same reason codes in sequence and thus it is important to preserve all the incoming information in such a way as to allow other modules to take their own action. In practice, it is legitimate for a module either to ignore a reason code, or to 'claim' it so as to prevent other modules from being offered it. These three circumstances are summarised below:

1. To refuse to deal with a reason code:
   Exit with all registers preserved using MOV PC,R14

2. To deal with a reason code and pass it on:
   Decode the reason code
   Preserve all registers
   Execute the appropriate software
   Restore registers
   Exit using MOV PC,R14

3. To 'claim' a reason code:
   Decode the reason code
   Execute the appropriate software
   Set R1=0 (indicating the call has been claimed)
   Exit using MOV PC,R14

Service Call Reason Codes

Let us now look at the meaning of each of the valid reason codes. There are quite a few of them, so don't feel you have to digest the whole lot on the first reading. The reason codes, which are passed to modules in register R1, are detailed below in numerical order. Where exit and entry conditions may be important in passing or returning information they are given.

R1=&00 Service Call was Claimed

Exit:

R1=0

This reason code is returned to indicate that the call has been claimed by the module. A module will never be called with this reason code as the OS will prevent further calling upon receiving it.

R1=&04 Unknown OS command

Entry:

R0=pointer to unknown command string

Exit success:

R1=0 (to claim the call)
R0=0 (to indicate no error)

Exit failure:

R1=(don't care)
R0=pointer to error block

The 'unknown command' reason code is provided for BBC MOS compatability – you do not need to respond to it since a more modern approach is available and described later.

If you can understand the command then you should execute the appropriate software and return by claiming the call with R1=0. If an error occurs you should return with R0 pointing to the error block.

R1=&06 Error Pending

Entry:

R0=pointer to error

This reason code is used to advise modules that an error has occurred but has not yet been dealt with by the error handler. To ensure that other modules find out too, you must not claim this call – simply return with all registers as they were received.

R1=&07 Unknown OS_Byte

Entry:

R2=OS_Byte number
R3=parameter 1
R4=parameter 2

Exit:

R1=0 to claim, otherwise preserve contents of R1

If your module provides extra OS_Byte functions they may be recognised and acted upon by means of this reason code. If OS_Byte number in R2 is one you provide, you should deal with it, and then claim the call with R1=0. Otherwise, return with all registers preserved so that other modules can have a go.

R1=&08 Unknown OS_Word

Entry:

R2=OS_Word number
R3=parameter 1

Exit:

R1=0 to claim, otherwise preserve as entry

This is the same as the above but for OS_Word functions. Note that OS_Word has just one parameter which is a pointer to a parameter block.

R1=&09 *HELP

Entry:

R0=pointer to *HELP command

The normal way of dealing with *HELP commands is through the help/command decoding table offset at the start of the module. This call is issued before *HELP decoding and should therefore only be claimed if you wish to completely replace the *HELP command (which is most unlikely and fairly rash).

R1=&0B Release FIQ vector

Immediately after the release of the FIQ vector (similar to NMI under BBC MOS) this call is issued to all other modules to advise them that the vector may be claimed (see below). For more information refer to the chapter on vectors (Chapter 20).

R1=&0C Claim FIQ vector

This call is used to advise modules that the vector is about to be claimed. For more information refer to the chapter on vectors (Chapter 20).

R1=&11 Memory Mapping Change

Entry:

R2=pointer to currently active module

Exit:

R1=0 to claim, otherwise preserve as entry

The OS issues this call when it needs to reorganise the memory map of the computer by altering the tables in the memory controller MEMC. By taking note of this call it is possible for your module to discover that it would be affected by such a reorganisation, and indeed may prevent such a reorganisation by claiming the call. On entry, R2 contains a pointer which, if within your module code, indicates that your module should claim the call.

R1=&12 Start Up Filing System

Entry:

R2=filing system number

When the user types a filing system startup command (eg, *NET or *ADFS) this call is issued to warn filing systems that they should shut down. Unless you are writing a filing system, which is outside the scope of this book, you do not need to take any notice of this call, which *must not* be claimed.

R1=&27 Machine Reset Warning

At the end of the machine reset sequence this call is issued to warn modules that a reset has occurred. This call *must not* be claimed.

R1=&28 Unknown *CONFIGURE Parameter

See R1=&29 below.

R1=&29 Unknown *STATUS Parameter

The above two reason codes are issued as requests to handle an unknown option to either *CONFIGURE or *STATUS. These were present on the earlier BBC and Master machines and are provided on the Archimedes for compatability. The OS provides a new system for handling configuration options which is described in a subsequent section.

R1=&2A Application About to Start

This call is offered around the modules to warn them that a new application is about to start up and use the application space. No parameters are passed on entry. A module can set R1=0 if it wants to prevent the new application from starting up! This may be useful, for example, if the application has data in it which is unsaved and would be destroyed by a new application taking over the application space.

R1=&40 Re-initialise Filing System

This call has no entry or exit parameters but signifies that the file manager has just re-initialised. If a module contains a file system, then it should respond to this call by adding its file system to those recognised by the manager. This can be done using OS_FSControl, with an 'add file system' reason code.

R1=&42 Translate File Type

This call is issued when the file manager is trying to convert an unrecognised file type number to a file type name. On entry, R2 contains the unrecognised file type number. If a module recognises this then it should return with the eight ASCII bytes of the file type name in R2 and R3. For more details see the section on OS_FSControl.

R1=&43 International Character Set Service

This call will be of interest to readers who are producing modules which provide international character sets and/or new keyboard alphabets. More details are contained in the PRM.

R1=&44 Advise Connected Keyboard Type

This call is offered around the modules to inform them of the type of keyboard which is in use on the machine. On entry R2=0 for the 'old style keyboard'. R2=1 for the A300-A400 machine keyboards. It may be of interest to modules which contain custom designed keyboard handlers.

R1=&45 Software Pre-reset

This call is issued to inform them that a software reset is about to occur. This means that the break key has been pressed and the Operating System is going to generate a reset. A software reset is distinct from a hardware reset which is caused by the RESET button being pressed.

R1=&46 Mode Change Warning

This call is to inform the modules that the Operating System has just changed to a new screen mode. It may be useful if a module provides screen related functions and needs to be aware of the properties of the current screen mode. It could update itself on receipt of this call by reading the new settings of the VDU variables.

Module Title String

Each module must have a title string to allow it to be identified by name when issuing *RM commands or via OS_Module. This offset points to the start of a zero-terminated string containing the module name, ideally adopting the Acorn standard of capitalising the first letter of each word. Spaces and control characters must be avoided.

Help String

This offset points to the zero-terminated string which will be displayed by *HELP MODULES etc. Spaces and tab characters (which tab to the next eight-character column) are allowed, but no other control characters should be included.

It is important to include a help string no matter how trivial the module. And to keep the output of *HELP MODULES tidy it is diplomatic to follow the Acorn standard, viz:

Module name <tab> v.vv (dd mm myyyy)

That is, the module name should be followed by one or more tabs to make it occupy 16 characters, the version number should be three decimal digits in the form v.vv and the date the software was released, if included, should be of the form 25 Feb 1965.

Help and Command Decoding Table

The help and command decoding table is the main mechanism that the OS employs to interact with a module and thus decide whether a particular command should be processed by that module. The table consists of a list of records, each of which contains a keyword and the associated information to allow that keyword to be processed – for example, the address of code to execute, the number of parameters expected and so forth. The keyword may be a command, a help subsection or (preferably) both.

By providing this table the module delegates the responsibility for decoding unknown commands to the OS, thus ensuring that command processing is consistent across all modules by preventing individual programmers from taking on the work themselves.

Decoding Table Layout

The format of the decoding table records is shown below. A record must appear for each keyword, whether it be a command or a help word.

ASCII keyword string, terminated by a zero byte
ALIGN to word boundary, before:
    Code offset for this keyword
    Parameter information word
    Offset to invalid syntax string, terminated by zero
    Offset to help text string, terminated by zero

Each record begins with the ASCII text string of the keyword terminated by a zero. We then align to a word boundary and insert the offset within the module at which may be found the code to interpret the keyword. If the keyword is for help only then we use a zero offset to indicate this. Next comes a word of information about the parameters (described below) and then an offset to a text message indicating that the user has employed an inappropriate syntax for the command. Finally, a word is included which is an offset to the string of help text to be generated when the keyword is used as a parameter to *HELP, eg:

```
*HELP  ThisKeyword
```

Keywords should only include alphabetic characters to be safe from the OS trying to decode them (believing them to be filenames or similar). The decoding process is case-insensitive, but for display purposes it is most elegant to continue the Acorn standard of capitalising the initial letters of each word.

The code offset points to the routine which will be called when the command is entered. If the offset contains zero the keyword can only be used as a *HELP parameter (and therefore, issuing the keyword as a command will result in a 'Bad command' error unless the command is recognised elsewhere). Otherwise, issuing the command '*KEYWORD' will cause the code to be executed with R0 pointing to the rest of the command line (stripped of leading spaces) and R1 containing the number of parameters discovered by OS_Cli (which uses spaces or double inverted commas as delimiters).

The parameter information word is divided up into its four constituent bytes, each of which has a different function:

Byte	Function
0	Minimum number of parameters allowed (0-255)
1	OS_GSTrans map (see below)
2	Maximum number of parameters allowed (0-255)
3	Flag byte

The minimum and maximum number of parameters fields are self-explanatory: if too few or too many parameters are supplied by the user the OS will display the syntax error message (pointed to by the appropriate offset given above).

OS_GSTrans map comprises a bit for each of the first eight parameters which indicates whether the OS should pass the parameter through OS_GSTrans before passing it on to the command execution software. This allows the burden of identifying and translation OS variables, for example, to be placed on the OS. When a bit is set the relevant parameter will be processed by GSTrans before receipt by the routine.

The flag byte is currently defined to contain three bits (the top three) which indicate special cases for commands. These are detailed below.

Bit 31

Setting this bit tells the OS that the command is specific to a particular filing system. OS_Cli will only acknowledge such commands if they are part of the current filing system and so different filing systems may support their

own specific versions of the same command without fear of being called at inappropriate times (ie, when they are not the current filing system).

Bit 30

When this bit is set the OS takes the keyword to be a parameter of the *CONFIGURE and *STATUS command (and only executes it where relevant). When the code is executed, R0 contains an indicator of the service required:

R0=0 The command was *CONFIGURE with no parameters so print a syntax string for the configuration commands the module recognises and return through R14.

R0=1 The command was *STATUS <keyword> so print the current configuration setting for the keyword and return through R14.

R0=<any other value>
For any other value the command was *CONFIGURE <keyword> and R0 points to the rest of the command line to allow the parameters to be extracted. The parameters should be decoded and the configuration bytes set as appropriate.

If the module detects a syntax error in the parameters it should return through R14 with the Overflow flag 'V' set and R0 containing a value indicating the type of error:

Value	Type of error
0	Bad configuration option
1	Numeric parameter required
2	Parameter too large
3	Too many parameters
>3	*CONFIGURE returns error code

Bit 29

Setting this bit indicates that the *HELP offset points to a piece of code instead of a text string. This allows *HELP information to vary (as it does, for example, under the ANFS when advising the user of their Econet station number). The code is entered with R0 pointing to a buffer area to use and R1 containing its size. The call is made with R2 containing either zero or a pointer to the remainder of the help string to be printed.

The invalid syntax string is printed by the OS whenever the parameters supplied by the user are outside the bounds specified in the minimum and maximum 'number of parameters' fields.

The help text string offset in the module header points to the text which will be printed if the keyword it refers to is appended after a *HELP command. The string is OS_PrettyPrinted, allowing tabs and hard spaces to be freely used within it to ensure tidy formatting on the screen.

SWI Chunk Base Number

The SWI chunk base number is the first of the optional SWI handling fields of the module header. It identifies the chunk of SWIs supported by the module, a chunk being defined to be 64 SWIs in total.

The OS uses this information to determine which module supports the unknown SWI which has been issued. If its number is in the range:

base to base+63

then this should be the module which deals with this SWI and the OS enters the appropriate piece of code through the next offset.

SWI Handling Code Offset

This is where the code to deal with SWIs is pointed to. It is up to the author of the module to ensure that the appropriate action is taken on receipt of the unknown SWI number.

On entry to the code, the ARM registers hold important information. R11 contains the SWI number within the given module (between 0 and 63), R12 points to the private word and R13 to the supervisor stack. The code is entered with interrupts disabled and they should be re-enabled if processing is going to take more than 20 microseconds (although the code must be able to cope with interrupts if you do re-enable them). To enable interrupts, use the following instructions:

```
MOV Rn, R14
TEQP Rn, #1<27  ;The IRQ bit is bit 27
```

The code should return from the entry point through R14 in the usual way, setting any relevant flags (eg, Overflow 'V' for errors) before doing so.

Note that the module does not have to deal with the 'X' prefix for SWIs, which allows optional error generation, since returning with 'V' set will cause the OS to generate the error automatically.

SWI Decoding Table

This offset points to a table to allow the translation of SWI names into numbers and vice versa (but see the note below). The format of the table is:

SWI group prefix (eg, 'Wimp')
Name of 0th SWI
Name of 1st SWI
.
.
Name of nth SWI
Zero to indicate end of list

where each of the text strings (the group prefix and the SWI names) is terminated by zero. The idea is that as many SWIs as possible should have their names included here so that the OS can deal with them itself. If the OS is trying to deal with a name which does not appear in this table or whose SWI number is greater than that of the last entry in this list, it calls the SWI decode code (below).

SWI Decode Code

On entry R0 indicates the required function and the remaining parameters are as follows:

If R0<0 then the OS wants to convert a text string (pointed to by R1) into a number. Return through R14 with R0 set to the SWI number in the range 0 to 63 or, if the SWI was unrecognised, return with R0<0.

If R0>=0 then the OS wants to convert the number in R0 into a text string. It supplies a pointer to its string processing buffer in R1, and R2 supplies the offset in that buffer at which it actually wants the string to be placed. The limit of the size of the buffer is in R3. The module is therefore expected to look up the string in its SWI decode table and place the string in the buffer (starting at R1+R2). R2 should be increased by the length of the string. Note that the zero terminator is added by the OS, so it does not need to be copied into the buffer.

A Note About SWI Translations

To summarise the effect of the two entry points above:

First, the SWI decoding table is used first by the OS to discern whether a particular SWI string can be recognised. If it does not appear in the table

(ie, its SWI number is greater than the last in the table) it calls the SWI decode code.

Secondly, the SWI decode code allows an application-specific mechanism for performing the translation to be implemented. It is up to you to get this right if you use it!

In general, it is better to use the SWI decode table fully and put zero in the SWI decode code offset so that it is not called. This keeps the behaviour of the OS consistent and reduces the chances of error, although it is a little less flexible.

OS_Module (SWI &1E)

The OS_Module SWI is the low-level call which performs the operations provided by the *RM series of OS commands. It also deals with other functions vital to the operation of the module system: for example, the claiming and releasing of RMA space.

OS_Module takes a reason code in R0 to activate each particular function, other parameters being specific to the individual function. Usually the OS will prevent you from applying this call to modules which it thinks are currently active – thus you may not 'kill' BASIC by calling that OS_Module function from BASIC itself.

When a filename is required (for example, to load or run a module) the file must have the correct type (&FFA) and respond to initialisation calls when loaded. Where the parameters refer to a module by name, the name is taken to be a string of upper or lower case characters which is terminated by a character with an ASCII code of less than 33. As ever, the Overflow flag 'V' will be set if an error occurs and R0 will point to an error block in the usual way. The exact cause of such an error is specific to each individual reason code.

R0=0 Run Module

This call is equivalent to *RMRUN, with R0 pointing to a module filename on entry. If the call succeeds it will not return.

R0=1 Load Module

This call is equivalent to *RMLOAD. On entry, R0 must point to a filename string. Space is allocated in the RMA and the module is loaded.

If a module of the same name already exists it will be overwritten. The Overflow flag 'V' will be set if an error occurs.

R0=2 Enter Module

This call is equivalent to *RMENTER Module, and allows a module to be made the current application. On entry, R1 must point to the module name string and R2 to the parameters to be passed on to the module. If the call is successful then User Mode is entered and the module started up.

R0=3 Re-initialise Module

This call is equivalent to *RMREINIT Module, resetting the module without having to reload it. On entry R1 must point to the module name string.

R0=4 Delete Module

This call attempts to remove the module and de-allocate any space the module may have claimed. On entry, R1 must point to the module name string.

R0=5 Get RMA Space

This call is passed on to the Heap Manager to get information about the RMA. It takes no entry parameters but returns, in R2, the size of the largest free space in the RMA and, in R3, the total free space.

R0=6 Claim RMA Space

This call allows the module to claim space in the RMA to use as workspace – performed by the Heap Manager. On entry, R3 should contain the size of the space you want and the call will return with R2 pointing to the start of the allocated space. If the space could not be allocated the Overflow flag 'V' will be set.

R0=7 Release RMA Space

This call has the reverse effect of the above, releasing space from the RMA heap by calling the Heap Manager. On entry, R2 should point to the block to release.

R0=8 Tidy RMA Space

This call polls each installed module and offers it a non-fatal finalisation call, allowing modules to de-allocate their space if they are able to do so. It then compacts the RMA space and reinitialises each module. Errors will be generated if any of the modules refuse to be finalised or reinitialised.

R0=9 Clear RMA Space

This call deletes each module in turn by calling them through their finalisation entry point. Errors will be generated if any of the modules refuses to be finalised.

R0=10 Create Module by Linking

It may sometimes be useful, especially during testing, to be able to install a module into the system which is not physically in the RMA space. This call allows the user to link a module, held anywhere in memory, into those already contained in the RMA. On entry R1 points to the address in memory at which the new module starts. On exit the module is an integral part of the system and behaves exactly the same as a RMA resident module. In future versions of the Operating System, it is probable that it will be required for the word immediately before the module to contain the module's length.

R0=11 Create Module by Copying

This call again installs a module which is not in the RMA. However, instead of linking the module into the system at its present address, it first copies it into the RMA space. The module is then initialised as if it had be loaded into the RMA from disc. The call is often useful when developing a module to copy a module, assembled into the application space, to the RMA for testing. On entry to the call, R1 points to the start address of the module and R2 must contain its length.

R0=12 Get Module Information

This call returns information about a given module. The information is that displayed by the command *MODULES, ie, the module's base address and the address of the start of its workspace in the RMA.

On entry R1 should contain '0' on the first call. It will then return the address of the first module in R1 and the address of its private word of workspace in R2. Subsequent calls to the routine, with R1 unchanged, will

return the same information on the second module, and then the third, and so on. When the last module is reached, R1 will be returned containing '0'.

R0=13 Extend RMA Block

This call allows a block of RMA workspace to be extended in size. The block will have been previously claimed by using OS_Module with R0=6. On entry to the routine, R2 points to the block of workspace (as returned when block was claimed). R3=amount to change the size of the RMA block by. On exit R2 points to the start address of the extended block. This may not be at the same address as the original block.

Printer Buffer Module

There follows a printer buffer module (listing 11.1) which will allow a variable size buffer to be used for background printing, (for example, whilst something else is going on).

```
 10 REM >List11/1
 20 REM by Nicholas van Someren
 30 REM Archimedes OS: A Dabhand Guide
 40 REM (c) Copyright AvS and NvS 1988
 50 :
 60 DIM code% 4000
 70 FOR pass%=4 TO 7 STEP 3
 80 REM Assemble for base address of zero, so all absolute
 90 REM addresses are relative addresses from start.
100 P%=0:O%=code%
110 [OPT pass%
120 EQUD 0                      ;Module cannot be run
130 EQUD initialise
140 EQUD finalise
150 EQUD service
160 EQUD title
170 EQUD helpstring
180 EQUD helptable
190 EQUD 0                      ;SWI chunk
200 EQUD 0                      ;SWI handler
210 EQUD 0                      ;SWI table
220 EQUD 0                      ;SWI code
230 :
240 .title                      ;Module title string
250 EQUS "PrinterBuffer"
260 EQUB 0
270 ALIGN
280 :
290 .helpstring                 ;Module help string
300 EQUS "Printer Buffer"+CHR$9+"1.00 (04 Dec 1987)"
310 EQUB 0
```

```
 320  ALIGN
 330  :
 340  .helptable
 350  EQUS "Buffer"              ;The first command
 360  EQUB 0
 370  ALIGN
 380  EQUD bufcommand            ;Address of code
 390  EQUD &00010001             ;Flags
 400  EQUD syntax                ;Syntax string
 410  EQUD bufhelp               ;Help string
 420  :
 430  EQUS "BufferSize"          ;The second command
 440  EQUB 0
 450  ALIGN
 460  EQUD sizecommand           ;Address of code
 470  EQUD &00010100             ;Flags
 480  EQUD sizesyntax            ;Syntax string
 490  EQUD sizehelp              ;Help string
 500  :
 510  EQUD 0                     ;End of command table
 520  :
 530  .bufhelp                   ;Help for *Buffer
 540  EQUS "*Buffer turns the extra printer buffer on and off."
 550  EQUB 13
 560  EQUB 10
 570  .syntax                    ;Syntax for *Buffer
 580  EQUS "Syntax: *Buffer <ON OFF>"
 590  EQUB 0
 600  :
 610  .sizehelp                  ;Help for *BufferSize
 620  EQUS "*BufferSize without a parameter gives the size of the
extra  print buffer."
 630  EQUB 13
 640  EQUS "With one parameter, the value is taken as the new
buffer size. "
 650 EQUB 13
 660  .sizesyntax                ;Syntax for *BufferSize
 670  EQUS "Syntax: *BufferSize [<size>]"
 680  EQUB 0
 690  ALIGN
 700  :
 710  .bufcommand                ;Code for *Buffer
 720  LDR R12,[R12]              ;Get private workspace addr
 730  LDRB R2,[R0],#1            ;Load first letter after command
 740  ORR R2,R2,#&20             ;Force lower case
 750  CMP R2,#ASC"o"             ;See if it is an 'o'
 760  BNE badonoff               ;If not, word is not On or Off
 770  LDRB R2,[R0],#1            ;Get next letter
 780  ORR R2,R2,#&20             ;Force lower case
 790  CMP R2,#ASC"n"             ;Is the next letter an 'n'?
 800  BEQ setup                  ;If so, do buffer 'setup' routine
 810  CMP R2,#ASC"f"             ;Is the letter an 'f'?
 820  BNE badonoff               ;If not, not On or Off
```

```
 830 LDRB R2,[R0],#1             ;Try the next letter
 840 ORR R2,R2,#&20              ;in lower case
 850 CMP R2,#ASC"f"              ;Is it an 'f'?
 860 BEQ setdown                 ;If it is, do buffer 'setdown'
 870 :
 880 .badonoff                   ;Cause an error
 890 ADR R0,bufneedonoff         ;Point R0 to error block
 900 ORR R14,R14,#1<<28          ;Set overflow flag for error
 910 MOVS PC,R14                 ;Return with flag set
 920 :
 930 .bufneedonoff               ;The error block for On/Off
 940 EQUD &00123456
 950 EQUS "*Buffer needs ON or OFF after it."
 960 EQUB 0
 970 ALIGN
 980 :
 990 .sizecommand                ;Code for the *BufferSize command
1000 STMFD R13!,{R14}            ;Stack the return address
1010 MOV R11,R12                 ;Keepcopy of private word addr
1020 LDR R12,[R12]               ;Load pointer from private word
1030 CMP R1,#0                   ;Were there zero parameters?
1040 BEQ tellsize                ;If so, just print out the size
1050 LDR R2,areweon              ;If not, see if the buffer is on
1060 CMP R2,#0                   ;Is the buffer 'off'?
1070 BNE changewhileon           ;If not, give an error
1080 MOV R1,R0                   ;R1 points to the command tail
1090 MOV R0,#10
1100 SWI "OS_ReadUnsigned"       ;Convert string to decimal value
1110 ADD R9,R2,#17               ;Add 17 to the required size
1120 LDR R10,[R12,#4]            ;Find old size of workspace
1130 SUB R3,R9,R10               ;Find the difference
1140 MOV R2,R12                  ;R2 point to workspace
1150 MOV R0,#13                  ;Extend by signed amount
1160 SWI "OS_Module"
1170 :
1180 ;NOTE - Arthur 1.20 has a bug in this code
1190 ;and does not set the V flag when an error occurs
1200 ;but does point to the error block with R0.
1210 :
1220 CMP R0,#13                  ;Test if R0 has changed
1230 BNE osmodulesbug            ;If it has, the bug has shown
1240 STR R2,[R11]                ;Store the new workspace address
1250                             ;in the private word
1260 STR R9,[R2,#4]              ;Store new workspace end pointer
1270 MOV R3,#&10                 ;Load the start of buffer pointer
1280 STR R3,[R2]                 ;and store at 'Start of buffer'
1290 STR R3,[R2,#8]              ;'Write in' pointer
1300 STR R3,[R2,#12]             ;and 'Read out' pointer
1310 LDMFD R13!,{PC}             ;Return
1320 :
1330 .tellsize                   ;User has asked for buffer size
1340 SWI "OS_WriteS"
1350 EQUS "The extra printer buffer is "
```

```
1360  EQUB 0
1370  ALIGN
1380  LDR R0,[R12,#4]             ;Get the size of workspace
1390  SUB R0,R0,#17               ;Subtract 17 byte overhead
1400  ADR R1,numbuffer            ;Point R1 to a string buffer
1410  MOV R2,#11                  ;(which is 11 bytes long)
1420  SWI "OS_ConvertCardinal4"   ;Convert size to a string
1430  SWI "OS_Write0"             ;Display size
1440  SWI "OS_WriteS"
1450  EQUS " bytes long."
1460  EQUB 13
1470  EQUB 10
1480  EQUB 0
1490  ALIGN
1500  LDMFD R13!,{PC}             ;Return
1510  :
1520  .osmodulesbug               ;Get around bug
1530  LDMFD R13!,{R14}            ;Load up return address
1540  ORR R14,R14,#1<<28          ;Set Overflow flag
1550  MOVS PC,R14                 ;Return with V set
1560  :
1570  .changewhileon              ;Say you can't change size
1580  LDMFD R13!,{R14}            ;Get return address
1590  ADR R0,changetext           ;Point to error block
1600  ORR R14,R14,#1<<28          ;Set V flag
1610  MOVS PC,R14                 ;Return
1620  :
1630  .changetext                 ;An error block
1640  EQUD &00123457
1650  EQUS "Can't change the buffer size when the buffer is on."
1660  EQUB 0
1670  ALIGN
1680  :
1690  .numbuffer                  ; Number/string conversion space
1700  EQUD 0
1710  EQUD 0
1720  EQUD 0
1730  :
1740  .initialise                 ;Module initialisation code
1750  STMFD R13!,{R14}            ;Push return address
1760  MOV R1,R12                  ;Keep a copy of private word addr
1770  LDR R12,[R12]               ;Load pointer to workspace
1780  CMP R12,#0                  ;Check if buffer is already
1790  LDMNEFD R13!,{PC}           ;installed and if so, return.
1800  MOV R0,#6                   ;R0=6 means 'claim space'
1810  MOV R3,#&4000               ;The default size is &4000 bytes
1820  SWI "OS_Module"             ;Claim some space
1830  STR R2,[R1]                 ;Store pointer in workspace
1840  MOV R12,R2                  ;Make R12 point to workspace
1850  STR R3,[R12,#4]             ;Store size as the end pointer
1860  MOV R3,#&10                 ;Load pointer to beginning
1870  STR R3,[R12]                ;Store as the beginning of buffer
1880  STR R3,[R12,#8]             ; the 'write in' point
```

```
1890  STR R3,[R12,#12]          ; and the 'read out' point
1900  LDMFD R13!,{PC}           ;Return
1910  :
1920  .finalise                 ;Module finalisation code
1930  STMFD R13!,{R14}          ;Stack return address
1940  LDR R12,[R12]             ;Get pointer to workspace
1950  BL setdown                ;Turn buffer off
1960  CMP R10,#0                ;See if it is a fatal shutdown
1970  MOV R0,#7                 ;Prepare to release space
1980  MOV R2,R12
1990  SWINE "OS_Module"         ;Release space if fatal shutdown
2000  LDMFD R13!,{PC}           ;Return
2010  :
2020  .service                  ;Module service code
2030  CMP R1,#&27               ;Look for post-reset service
2040  MOVNE PC,R14              ;If not, return
2050  STMFD R13!,{R0}           ;Buffer is 'off' after reset
2060  MOV R0,#0
2070  STR R0,areweon
2080  LDMFD R13!,{R0}           ;Preserve registers
2090  MOV PC,R14                ;Return
2100  :
2110  .areweon                  ;'Is the buffer on?' flag
2120  EQUD 0
2130  :
2140  ;+0 is start of buffer from 0
2150  ;+4 is end of buffer from 0
2160  ;+8 is the point for insertion
2170  ;+12 is the point for removal
2180  :
2190  .nextval                  ;Increment R4 with wrap around
2200  LDR R5,[R12,#4]           ;Find the end of the buffer
2210  ADD R4,R4,#1              ;Increment R4
2220  CMP R4,R5                 ;See if R4 points off the end
2230  LDREQ R4,[R12]            ;If so, load up start point
2240  MOVS PC,R14               ;Return, returning all flags
2250  :
2260  .isempty                  ;Test if buffer is empty
2270  LDR R4,[R12,#8]           ;Look at 'write in' point
2280  LDR R5,[R12,#12]          ; and 'read out' point
2290  CMP R4,R5                 ;and compare them
2300  MOV PC,R14                ;Return
2310  :
2320  .isfull                   ;Test if buffer is full
2330  STMFD R13!,{R14}          ;Stack return address
2340  LDR R4,[R12,#8]           ;Get insertion address
2350  BL nextval                ;What will it be after next ?
2360  LDR R5,[R12,#12]          ;Look at removal address
2370  CMP R4,R5                 ;Will insertion catch up?
2380  LDMFD R13!,{PC}           ;Return
2390  :
2400  .topush                   ;Push a byte into buffer
2410  STMFD R13!,{R14}          ;Stack return address
```

```
2420  BL isfull                 ;See if buffer is full
2430  LDMEQFD R13!,{R14}        ;If it is, get the return address
2440  ORREQS PC,R14,#1<<29      ;and return with C set
2450  LDR R4,[R12,#8]           ;Otherwise, load insertion addr
2460  STRB R0,[R12,R4]          ;Store the byte
2470  BL nextval                ;Increment the 'write in' pointer
2480  STR R4,[R12,#8]           ;Store pointer back
2490  LDMFD R13!,{R14}          ;Load return address
2500  BICS PC,R14,#1<<29        ;Return, ensuring C is clear
2510  :
2520  .topull                   ;Pull a byte from buffer
2530  STMFD R13!,{R14}          ;Stack return address
2540  BL isempty                ;See if buffer empty
2550  LDMEQFD R13!,{R14}        ;If it is, load the return addr
2560  ORREQS PC,R14,#1<<29      ;Return with C set
2570  LDR R4,[R12,#12]          ;Otherwise, load the removal addr
2580  LDRB R0,[R12,R4]          ;Load next byte into R0
2590  MOV R2,R0                 ;Duplicate it into R2
2600  TST R6,#1<<28             ;Check for examine only
2610  BLEQ nextval              ;If not, increment removal addr
2620  STR R4,[R12,#12]          ;and write it back
2630  LDMFD R13!,{R14}          ;Load return address
2640  BICS PC,R14,#1<<29        ;Return with C clear
2650  :
2660  .toaltflush              ;Flush buffer if Alt is pressed
2670  STMFD R13!,{R0,R1,R2,R14} ;Push enough regs to do OS_Byte
2680  MOV R0,#&81               ;Call OS_Byte &81, check for Alt,
2690  MOV R1,#&FD               ;(that is, check for INKEY(-3))
2700  MOV R2,#&FF
2710  SWI "OS_Byte"
2720  CMP R1,#&FF               ;Was Alt pressed?
2730  LDMFD R13!,{R0,R1,R2,R14} ;Pull back registers anyway
2740  MOVNE PC,R14              ;Return if Alt not pressed
2750  LDR R4,[R12]              ;The start of the buffer
2760  STR R4,[R12,#8]           ;becomes the insertion address
2770  STR R4,[R12,#12]          ;and the removal address
2780  MOV PC,R14                ;Return
2790  :
2800  .tocountpurge             ;Perform count and purge
2810  TST R6,#1<<28             ;Look at the V flag
2820  BNE toaltflush            ;If set, try to flush buffer
2830  STMFD R13!,{R14}          ;Otherwise, stack the return addr
2840  TSTP R6,R6                ;Set flags as they are in R6
2850  LDRCS R4,[R12,#8]         ;If carry set, R4 is insert point
2860  BLCS nextval              ;which is then incremented
2870  LDRCS R5,[R12,#12]        ;and R5 is remove point
2880  LDRCC R5,[R12,#8]         ;If carry clear, R4=remove point
2890  LDRCC R4,[R12,#12]        ;and R5 is insert point
2900  SUBS R1,R5,R4             ;Take the diference
2910  BHI posspace              ;If diference positive, it's OK
2920  LDR R4,[R12]              ;Otherwise, get the buffer start
2930  LDR R5,[R12,#4]           ;and the end of the buffer
2940  ADD R1,R1,R5              ;Add in one
```

```
2950  SUB R1,R1,R4                ;and take out the other
2960  .posspace                   ;The value in R1 is now positive
2970  MOV R2,R1,LSR #8            ;Put the top 3 bytes into R2
2980  LDMFD R13!,{PC}             ;Return
2990  :
3000  .myinsv                     ;Entry in the insert vector list
3010  CMP R1,#3                   ;Check for printer buffer
3020  MOVNES PC,R14               ;If not, return
3030  STMFD R13!,{R0,R1,R4,R5}    ;Push some registers
3040  BL topush                   ;Do the push
3050  LDMFD R13!,{R0,R1,R4,R5,PC}  ;Return with registers
3060  :
3070  .myremv                     ;Entry in remove vector list
3080  STMFD R13!,{R6}             ;Stack R6
3090  MOV R6,PC                   ;and put a copy of the PC in it
3100  CMP R1,#3                   ;Check for printer buffer
3110  LDMNEFD R13!,{R6}           ;Restore R6 if not printer buffer
3120  MOVNES PC,R14               ;Return if not printer buffer
3130  STMFD R13!,{R1,R4,R5}       ;Stack up some registers
3140  BL topull                   ;Do the pull
3150  LDMFD R13!,{R1,R4,R5,R6,PC}  ;Return with all registers
3160  :
3170  .mycnpv                     ;Entry in count/purge vector list
3180  STMFD R13!,{R6}             ;Preserve R6
3190  MOV R6,PC                   ;Take a copy of PC
3200  CMP R1,#3                   ;Test for printer buffer
3210  LDMNEFD R13!,{R6}           ;Restore R6 if not printer buffer
3220  MOVNES PC,R14               ;Return if not printer buffer
3230  STMFD R13!,{R4,R5}          ;Stack more registers
3240  BL tocountpurge             ;Do the count or purge
3250  LDMFD R13!,{R4,R5,R6,PC}    ;Return with all registers
3260  :
3270  .setup                      ;Routine to install buffer
3280  STMFD R13!,{R14}            ;Stack return address
3290  LDR R0,areweon              ;Load the flag
3300  CMP R0,#0                   ;Compare with zero
3310  LDMNEFD R13!,{PC}           ;If not, already installed
3320  MVN R0,#0                   ;Otherwise, set the flag
3330  STR R0,areweon              ;and store it.
3340  MOV R2,R12                  ;R12 wanted on entry to vector
3350  LDR R0,[R12]                ;Empty buffer
3360  STR R0,[R12,#8]
3370  STR R0,[R12,#12]
3380  MOV R0,#&14                 ;Buffer insert vector
3390  ADR R1,myinsv               ;Use address of new routine
3400  SWI "OS_Claim"              ;and claim vector
3410  MOV R0,#&15                 ;Buffer remove vector
3420  ADR R1,myremv
3430  SWI "OS_Claim"
3440  MOV R0,#&16                 ;Buffer count/purge vector
3450  ADR R1,mycnpv
3460  SWI "OS_Claim"
3470  LDMFD R13!,{PC}             ;Return
```

```
3480  :
3490  .setdown                  ;Routine to shut down buffer
3500  STMFD R13!,{R14}          ;Stack return address
3510  LDR R0,areweon            ;Load on/off flag
3520  CMP R0,#0                 ;Test if buffer is on
3530  LDMEQFD R13!,{PC}         ;Don't turn off if off
3540  MOV R0,#0                 ;Set flag as being off
3550  STR R0,areweon            ;Save the flag
3560  MOV R2,R12                ;Need installation value of R2
3570  MOV R0,#&14               ;Buffer insert vector
3580  ADR R1,myinsv             ;Extra entry
3590  SWI "OS_Release"          ;Release
3600  MOV R0,#&15               ;Buffer remove vector
3610  ADR R1,myremv
3620  SWI "OS_Release"
3630  MOV R0,#&16               ;Buffer count/purge vector
3640  ADR R1,mycnpv
3650  SWI "OS_Release"
3660  LDMFD R13!,{PC}           ;Return
3670  ]:NEXT pass%
3680  OSCLI"SAVE BufMod "+STR$~code%+"+"+STR$~P%
3690  OSCLI"SETTYPE BUFMOD FFA"
3700  PRINT"Printer buffer module saved as 'BufMod'"
3710  END
```

Listing 11.1. Printer Buffer Module.

12 : Writing Applications

The Operating System provides support for two essentially different types of software: applications and utilities. Typically an application will be a major self-contained program like a word processor or a game. Such a program would run in the ARM's User Mode to maximise protection against bugs. A utility would usually be an extension to the operating system and therefore run in ARM's Supervisor Mode, with all the control (and lack of security) which this affords.

Mechanisms are built in to provide for and formalise the execution of both applications and utilities. There are several reasons for the Operating System wanting to keep track of which is the current application and where in memory it is, mainly to do with the protection of system workspace and workspace for other programs. As a result, in order to start an application the user must tell the Operating System through one of the prescribed channels.

Starting Applications

One way to start an application is to enter the *GO command at the Supervisor command line or through the OS_CLI SWI. The *GO command takes at least one parameter, the first parameter being the memory address of the machine-code that needs to be executed as an application. The rest of the parameters in the command are passed on to the application for its specific use.

The other way to start an application from the command line is to use the *RUN command on a file. When a machine code file is run which has no file type set, it is run as an application in a manner equivalent to *LOADing the file at the load address and then issuing a *GO to the execution address. An example of an application started in this way is the TWIN text editor used to write this book.

Applications have a certain number of privileges, notably the right to use the lion's share of the RAM in the machine. Application memory is deemed to start at &8000 and extend up to limits defined by the other applications in use (if any). In order to find out what resources are at its disposal the

application may issue a number of SWIs. The most important of these and the one that all applications will need to use is OS_GetEnv.

OS_GetEnv (SWI &10)

This SWI returns a number of important pieces of information about the 'environment' in which an application is running. It takes no parameters but returns three pointers in R0, R1 and R2.

R0 points to the text of the command line used to start the application running. This string will have been modified by aliases, by parameter substitution and, if *GO was used to start the application, the address after *GO will have been removed. Thus the string will always be a command followed by a parameter list for the application. Applications which can take a parameter such as a file name (eg, text editors) may therefore examine this string for filenames etc.

R1 points to the first byte of memory that is not available to the application, ie, if R1 contains &60000 then the application may use the memory from &8000, the start of application RAM, to &5FFFF but may not use the bytes from &60000 upwards. If the application is going to run for more than a few instants, or if it is going to build a long stack, it should set R13 (the usual stack pointer) to this value so that the stack will extend downwards into the application space from the top.

R2 points to a five byte area containing the absolute time at which the application was started. This is in the standard format of the number of centiseconds since the start of this century and can be converted to a displayable string using the SWI OS_ConvertStandardDateAndTime.

Alternative Ways of Starting Applications

There are two further ways in which a piece of machine code can be started as an application. Firstly, machine code intended to be run at &8000 can be given file type of &FF8 (Absolute) using *SETTYPE and the the file can be *RUN. Alternatively the code may be in a module, as BASIC V is, and be executed when that module is started. To demonstrate the use of OS_GetEnv in this situation enter the following few lines at the BASIC prompt:

```
SYS "OS_GetEnv" TO A%,B%
SYS "OS_Write0",A%
PRINT ~B%,~HIMEM
```

The BASIC module is the current application so you can see the string that started BASIC running and the workspace limit, along with BASIC's pseudo-variable HIMEM which defines the upper limit of memory for BASIC.

In order to allow a module to start up as an application, it must respond to a *Command defined in its command table which activates the module as an application by using OS_Module with R0=2. Some example code to do this can be seen in the Shell module listed below.

Temporarily Running Another Application

A particularly useful thing to be able to do from inside an application is to temporarily run another application. If, for example, you are working within a text editor and you need to perform some quick calculation, it would be a pity if you had to save your work and exit completely. Another example is wishing to use the *COPY command in its 'quick' mode where it behaves as if it were an application. Note that, being able to do this is not multi-tasking since only one application is running at a time, but it is a major timesaver nevertheless. In order for applications to be nested in this way we need to be able to exit from them just as neatly as we enter them, preserving important workspace and so forth as we go.

All the methods of starting applications examined above are effectively 'permanent', ie, once an application is running it is the only one that the Operating System takes care of. To leave that application an OS_Exit SWI should be issued, returning control back to an Operating System handler routine designed for this purpose. By default this handler returns control to the Supervisor (with its '*' prompt). However, an application started previously can set the address to which control will be returned by the exit handler using the SWI OS_ChangeEnvironment. This allows control to be returned to it later, providing the basis for a system of nested application calls.

To change the OS_Exit handler return address R0 should contain 11 and SWI OS_ChangeEnvironment should be called. On return R1 will contain the new return address. On return from this SWI R1 will contain the old return address which should be stored safely for use when OS_Exiting from your application, so that a string of applications may be nested.

Top Down Applications

If one simply sets the return address to some location within the current application before exiting, there is a good chance that the new application will blindly trample over all of the available workspace, unaware that

other applications may wish to preserve it. This is, of course, the situation where just one application is in use at a time, but is insufficient to provide for nested applications. Instead, an application which is designed to use workspace from the top down can protect itself from other applications by lowering the upper workspace limit which will be returned to other applications, thus rendering itself 'invisible'.

OS_ChangeEnvironment provides a call for this purpose: when entered with R0=0 it will set the apparent top of workspace to the value in R1. By writing an application in such a way that it steals workspace from the top of memory and sets the new workspace limit to just below the lowest location, it uses other applications which will not conflict.

An example of this approach is detailed later in the "Shell" module. When this module is entered as an application it examines the upper workspace limit and reduces it by 256 bytes. It then saves the previous values of the environment parameters and goes about its business of repeatedly reading lines entered at the keyboard into a buffer and passing them to the command line interpreter. It continues to do this until a blank line is entered, at which point it restores the old environment parameters and exits. Some care needs to be taken, since on returning with OS_Exit the contents of registers are undefined.

Although the Shell module is only a demonstration, not too useful in itself, it is a fully-fledged application. You might like to use it as a template for producing your own top down applications.

The TWIN Text Editor

Acorn's TWIN editor uses the top down approach in an interesting (if a little naughty) manner. TWIN is designed to be loaded somewhere in the middle of the application workspace; the memory above the program code is used to store the text being edited. The memory below the start of the program is made available to other applications that can be started from the command line (eg, *COPY). TWIN is not altogether satisfactory in this respect as the position at which it is loaded is fixed (by its default load address) and can only be altered by means of system calls or a special utility. A much better solution to the problem, and one used by the Shell program, is to have the program code resident in a module (and thus always accessible) and, when the program is started as an application, to take workspace from the top of the available RAM as discussed above. When other applications are started from the command line they may use workspace up to the new limit and this process may be repeated until no more workspace is available.

An alternative way of allocating workspace would be to allow applications to claim workspace from within the RMA. This has the virtue that the size of the workspace may be dynamic, because the Operating System allows the memory in the workspace to be consumed and replaced at will using memory management routines; these are discussed in the next section.

Memory Management

All applications require memory of one form or another in which to operate. When programming in BASIC it is not necessary to concern oneself with issues of memory management because the BASIC interpreter does the work for you. However, if you intend to write stand-alone applications of your own, particularly in assembler, it is important to be aware of the facilities provided for managing memory. These issues are made all the more important when more than one application is running under a multi-tasking OS such as RISC OS.

A number of routines are provided to allow applications to do their own memory management. They are all based on a standard memory concept, known as a 'heap', from which chunks of memory may be claimed and released by applications with the assistance of the OS.

ARM Memory Structure

The ARM CPU has a possible address space of 64Mbs, of which only the lower 32Mbs is relevant in Archimedes because this is the space in which RAM is seen to exist. The ARM and MEMC chips contrive to make this 32Mbs space (known as the 'logical' address space) available to applications in User Mode by translating references to it through MEMC's internal tables. This allows the OS to decide which sections of the logical address space are actually present as 'physical' memory, and this forms the basis of a segmented 'virtual' memory system.

Physical memory is divided up into 128 'pages' whose size varies according to the amount of memory actually fitted to the machine: 300 Series computers have pages of 8k while 400 series machines have pages of 32k. Blocks of pages are allocated as the workspace for particular OS functions according to the configuration settings used. See the chapter on the ARM support chips for more detailed information.

Heap Management Software and SWIs

The OS Heap Manager provides routines which allocate and de-allocate chunks of memory from a chosen area of memory. This mechanism is used by the OS to deal with memory management within the RMA amongst others. User applications can make use of the Heap Manager's facilities by providing it with information about the size of a block of memory and then calling it for allocations and de-allocations.

The Heap Manager keeps track of the component chunks of the heap by means of heap description blocks which consist of four word units. To start using the Heap Manager you need some memory; if you are writing an application you may use the application workspace or, if you are writing a module, you may claim memory from the RMA using OS_Module with R0=6 (see the chapter on Modules). The pointer to the start of free memory (perhaps returned by OS_Module) should be handed to the Heap Manager for it to place its heap descriptor block at the start. After the heap has been initialised (see below) other calls may then be made to the Heap Manager to allocate and de-allocate blocks.

One SWI is used to deal with the Heap Manager – OS_Heap. On entry, it takes a reason code in R0 and appropriate parameters in other registers, returning with results in registers and the Overflow flag 'V' set if any error occurred.

OS_Heap (SWI &1D)

R0=0 Initialise Heap

This call must be made once before any use is made of other calls to OS_Heap. On entry, R1 should point to the start of the area of memory to be used as a heap (the first four words of which will be used as the heap description block) with R3 containing its size (in bytes). Both of these values must be word-aligned and they may not exceed 32Mb and 16Mb respectively.

R0=1 Describe Heap

This call gives information about the state of the heap – its largest available block and the total amount of free space. On entry, R0 must point to the heap description block. The results returned are the size of the largest available block in R2 and the total free space in R3. The Overflow flag 'V' is set if the heap description block was invalid.

R0=2 Claim Heap Block

This call allocates a block from the heap if sufficient space exists. On entry, R1 must point to the heap description block and R3 must contain the desired size of block. On return, either R2 is a pointer to the allocated block (if sufficient space was free) or the Overflow flag 'V' is set (if there was insufficient space).

R0=3 Release Heap Block

This call releases a block of the heap if the supplied pointer is valid. On entry, R1 must point to the heap description block and R2 to the block to be de-allocated. No results are returned but the Overflow flag 'V' will be set if either of the block pointers was invalid.

R0=4 Extend Heap Block

This call allows you to increase or decrease the size of a currently allocated block. On entry, R1 points to the heap and R2 to the block itself with R3 containing the signed 32-bit number by which the size of the block should be altered. On return, R2 contains a pointer to the new block. Note: since the Heap Manager may have had to allocate new space to accommodate your request, the pointer to the block may have changed – you must take note of any such change yourself.

There is a bug in Arthur 1.20 which means that this call does not always work. Sometimes it fails to reallocate the block correctly if the block needs to be moved within the heap. Unfortunately, no detailed information was available at the time of writing.

R0=5 Extend the Heap

Contrary to the description in the Programmer's Reference Manual, this call is *not implemented* in Arthur 1.20.

Listing 12.1 at the end of this section illustrates the OS_Heap SWI. A heap is initialised and three blocks of memory are claimed from it. The procedure PROCdescribeheap calls OS_Heap with R0=1 in order to extract details about the heap.

OS_ValidateAddress (SWI &3A)

One SWI is provided which allows you to establish whether a range of memory locations is accessible in User Mode (ie, that MEMC has the specified logical memory range paged into physical memory).

On entry, R0 and R1 contain the lower and upper limits of the address range to be checked. The call returns with the Carry flag 'C' clear if the address range is currently paged into physical RAM and 'C' set if not. Listing 12.3 at the end of this section illustrates the use of OS_ValidateAddress.

There is a bug in Arthur 1.20 which causes references to some addresses below screen memory to be marked as valid when, in fact, they are not paged into physical memory – beware!

General Guidelines on Compatibility

Acorn documentation suggests a number of guidelines which, if followed, will maximise the chances that existing software will be easily portable onto future versions and releases of the OS, so that programs written will run under Arthur 1.2, RISC OS and any future release. It is important to remember that this is one of the reasons we have an Operating System at all – to improve portability. Common sense is the best guide here, so SWIs should be used instead of calling ROM addresses, hardware should *never* be written to directly and so on. Many of the SWIs the OS provides are present to allow access to internal information – be sure to use them. It is a good idea to take advantage of the WIMP facilities when writing applications, as this is clearly Acorn's intention in providing them.

Another aspect of portability, which is critical, is to allow software to be easily installed on hard discs. This raises a number of issues to do with directory structures, in particular assumptions about the root directory. Firstly, files should never be referred to by their full pathname, eg, $.MyApp.RunCode. If they are, moving the program onto a hard disc means duplicating the whole directory structure.

Instead, utilities used by your application should be kept in a library directory in the root, resources such as fonts should be kept in some suitably named directory in the root, eg, $.Fonts, and anything else should grow from a sub-directory whose name is the same as that of the application. You can then use the *URD command to set this application directory as the User Root Directory during initialisation. By ensuring that the application

only ever refers to its own directory using the URD symbol '&' we can keep the application easily portable.

To get things started there could be a text file in the library used to start the application called, say, "StartApp" and containing:

*URD $.MyAppDir	To set up the User Root
*SET Font$Prefix $.Fonts	To set up resources (ie, the fonts)
*RUN &.RunCode	To start the application

To move the application to another system with a hard disc, all that needs to be done is to copy the application directory tree structure, the contents of the library and the contents of the resource directories over to the hard disc root. Then, simply editing the the first line of the startup file will install the new application.

The Shell Module - Source Code

```
 10  REM >List12/1
 20  DIM code% 511
 30  FORpass%=0TO3STEP3
 40  P%=0:O%=code%
 50  [OPT pass%+4
 60  EQUD start                  ;Entry point as an application
 70  EQUD 0
 80  EQUD 0
 90  EQUD 0
100  EQUD title
110  EQUD help
120  EQUD helptable
130  EQUD 0
140  EQUD 0
150  EQUD 0
160  EQUD 0
170
180  .title
190  EQUS "ShellModule"
200  EQUB 0
210  .help
220  EQUS "Shell Module"
230  EQUB 9
240  EQUS "1.00 (05 Jul 1988)"
250  EQUB 0
260  ALIGN
270
280  .helptable
290  EQUS "Shell"
300  EQUB 0
310  ALIGN
320  EQUD doshell                ;Entry point for the command
```

```
330 EQUD 0
340 EQUD shellsyntax
350 EQUD shellhelp
360 EQUD 0
370
380 .shellhelp
390 EQUS "*Shell starts up the shell application."
400 EQUB 13
410 .shellsyntax
420 EQUS "Syntax : *Shell"
430 EQUB 0
440 ALIGN
450
460 .doshell
470 ADR R1,title                ;Pass the name of the module
480 MOV R2,R0                   ;the command tail
490 MOV R0,#2                   ;R0=2 for start module app.
500 SWI "OS_Module"             ;Do the SWI
510
520 .start
530 SWI "OS_GetEnv"             ;Find the RAM limit
540 SUB R12,R1,#&100            ;Move RAM limit down
550 MOV R0,#0                   ;Indicate change in RAM limit
560 MOV R1,R12                  ;new RAM limit
570 SWI "OS_ChangeEnvironment"
580 STR R1,[R12,#8]             ;Store old RAM limit
590 MOV R0,#11                  ;Indicate change in EXIT handler
600 ADR R1,exitback             ;New Exit address
610 MOV R2,R12                  ;Value returned in R12 on exit
620 SWI "OS_ChangeEnvironment"
630 STMIA R12,{R1,R2}           ;and store the old lot
640 .loopback
650 SWI &12A                    ;Print a star
660 ADD R0,R12,#32              ;find the line buffer
670 MOV R1,#127                 ;line length
680 MOV R2,#32
690 MOV R3,#255
700 SWI "OS_ReadLine"           ;read a line
710 CMP R1,#0                   ;Is it of zero length ?
720 BEQ exit                    ;If so to do the Exit
730 SWI "OS_CLI"                ;Otherwise execute the command
740 .exitback
750 B loopback                  ;and loop back
760 .exit
770 MOV R0,#0                   ;To reset the RAM limit
780 LDR R1,[R12,#8]             ;get old RAM limit
790 SWI "OS_ChangeEnvironment" ;and set it
800 MOV R0,#11                  ;Resetting EXIT handler
810 LDMIA R12,{R1,R2}           ;get old handler
820 SWI "OS_ChangeEnvironment" ;and set it
830 MOV R0,#0
840 MOV R1,#0
850 MOV R2,#0
```

```
 860  SWI "OS_Exit"                ;And exit
 870  :
 880  ]
 890  NEXT
 900  OSCLI"Save ShellMod "+STR$~code%+"+"+STR$~P%
 910  OSCLI"SetType ShellMod FFA"
```

Listing 12.1. The Shell source.

```
  10  REM >List12/2
  20  REM by Nicholas van Someren
  30  REM Archimedes OS: A Dabhand Guide
  40  REM (c) Copyright AvS and NvS 1988
  50  :
  60  REM Reserve plenty of space for the heap.
  70  REM and initialise it.
  80  :
  90  DIM Heap &10000
 100  SYS "OS_Heap",0,Heap,,&10000
 110  PROCdescribeheap
 120  :
 130  REM Get three blocks from the heap.
 140  :
 150  SYS "OS_Heap",2,Heap,,&400 TO ,,block1k
 160  SYS "OS_Heap",2,Heap,,&800 TO ,,block2k
 170  SYS "OS_Heap",2,Heap,,&1000 TO ,,block4k
 180  PRINT"Blocks of 1,2 and 4k have been allocated at
&";~block1k;",&";~block2k;" and &";~block4k;" respectively."
 190  PROCdescribeheap
 200  :
 210  REM Return the 1k block to the heap and extend
 220  REM the 4k block by 1k.
 230  :
 240  SYS "OS_Heap",3,Heap,block1k
 250  SYS "OS_Heap",4,Heap,block4k,&400
 260  PRINT"The 1k block has been deallocated, and the 4k block
extended by 1k"
 270  PROCdescribeheap
 280  :
 290  REM Note: extending the 2k block instead will reveal
 300  REM a bug in Arthur 1.20.
 310  REM SYS "OS_Heap",5 does not work at all!
 320  END
 330  :
 340  DEF PROCdescribeheap
 350  SYS "OS_Heap",1,Heap TO ,,largeblock,space
```

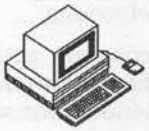
```
360 PRINT'"The largest block is &";~largeblock;" bytes long."
370 PRINT"There are &";~space;" bytes free."'
380 ENDPROC
```

Listing 12.2. The Heap.

```
 10 REM >List12/3
 20 REM by Nicholas van Someren
 30 REM Archimedes OS: A Dabhand Guide
 40 REM (c) Copyright AvS and NvS 1988
 50 :
 60 REM Define step size as &2000, start address as 0,
 70 REM total memory found as 0 and validate start address.
 80 :
 90 chunk%=&2000
100 addr%=0
110 total%=0
120 SYS "OS_ValidateAddress",addr%,addr% TO ;C%
130 :
140 REM As long as addr% is in user area and addr% is
150 REM not valid, increment the address and try again.
160 :
170 WHILE addr%<&2000000
180 WHILE C% AND 2
190 addr%+=chunk%
200 SYS "OS_ValidateAddress",addr%,addr% TO ;C%
210 ENDWHILE
220 :
230 REM Once a valid address is found, find how long the
240 REM valid block of memory is.
250 :
260 PRINT"Block from &";~addr%;
270 start%=addr%
280 WHILE (C% AND 2)=0
290 addr%+=chunk%
300 SYS "OS_ValidateAddress",addr%,addr% TO ;C%
310 ENDWHILE
320 PRINT" to &";~addr%-1;" that is &";~addr%-start%;" bytes
long"
330 total%+=(addr%-start%)
340 ENDWHILE
350 :
360 REM Once address is over &2000000, print out the total.
370 :
380 PRINT ;total%/1024;"k bytes asigned."
390 PRINT"Note : There is a bug in OS_ValidateAddress in Arthur
1.20 - "
400 PRINT"the last block will be incorrect."
410 END
```

Listing 12.3. Validate Address

13 : The Window Manager



This chapter is devoted to the intricacies of the Window Manager, sometimes known as the WIMP system, and how to use and program with it. The Window Manager is used extensively by the Desktop, so if you are not already familiar with the Desktop you might find it useful to experiment with it before proceeding.

The Window Manager provides a rich set of facilities which allows programmers to produce applications with a consistent style of user interface. This consistency has benefits both for non-technical users (because they have less to remember) and for programmers (because it simplifies programming). Furthermore, adopting Acorn's window standard will ease the movement of applications up to future versions of the OS. This is of particular relevance in allowing several applications to be used concurrently under Operating Systems such as RISC OS.

What's On Offer?

The Window Manager co-ordinates the user interface aspects which have recently become fashionable throughout the computer world, namely: windows, icons, mice and pull-down menus (hence the acronym WIMP). This approach to user interface design was popularised by the Apple Macintosh, though its origins were in Xerox's work with the programming environment known as Smalltalk.

The idea is that different pieces of work appear on the display as a series of windows, analogous to sheets of paper. Their size, shape and position may be altered by the user to suit the task currently in hand. This allows each piece of work to be imagined as occupying as much display space as necessary, with some smaller part of it actually being visible according to the prevailing size of the window. By re-sizing and re-positioning windows the user can change the focus of their attention as desired. Such effects are achieved through the use of the mouse, which allows positions on the display to be indicated simply by 'pointing'. The three mouse buttons are used to select and adjust various controls by 'clicking' on display 'buttons'. These buttons are defined to have specific meanings under the Window Manager: the left-hand button is for 'selection', the middle button

for pulling down 'menus' and the right-hand button is for 'adjusting' selections.

Icons are application-specific graphical objects, usually designed so that they represent the function they affect, which may be positioned within a window for selection with the mouse.

'Menus' of functions may be provided that allow software to be controlled by pointing to a menu entry rather than typing at the keyboard, and this mechanism can be extended more or less ad infinitum by the use of sub-menus from the first menu and so on. The use of menus ensures that the user can only make valid selections (since the menus won't contain invalid ones) and thus greatly reduces the scope for user error.

The Structure of Windows

A window consists of an area containing the work (known as the 'Physical Work Area' or PWA) bounded by a number of controls for the window. In the centre of the top of the window is a 'title bar' which contains the textual name attached to the window. A window may be 'dragged' around the display by holding down the select button, while the pointer is within the title bar, and then moving the pointer elsewhere, releasing the button when the desired destination is reached.

At the top left corner of most windows are two buttons: 'go to back' which moves the window behind any other windows on the screen; and 'close' which removes the window completely. At the top right corner is the 'toggle' button which causes the window to expand to its full size and move to the front (if it is not already there).

In the bottom right-hand corner of most windows is a 'stretch' button – dragging it allows the overall size of the window to be increased and decreased within limits set by the programmer. The maximum possible size is the same as that achieved by clicking on the full size button.

Along the right-hand side and/or the bottom edge of most windows are 'scroll bars'. When the window is not as large as the maximum working area it is possible to scroll the PWA over the whole extent of the work (known as the 'Logical Work Area' or LWA) to make any portion visible. At each extreme of the scroll bars are arrow buttons which scroll the PWA in the appropriate direction by a small amount. Within the scroll bars are shaded boxes whose size indicate how much of the LWA is visible within the PWA. Their position indicates which part (rather than how much) is visible. You can drag these boxes within the scroll bars to make large changes to the visible portion; repeated clicking on the arrows gets a little tedious.

Window Manager Co-ordinate Systems

The main part of any window is the Physical Work Area (PWA) of that window. This is bounded by two co-ordinate pairs which define its position on the display – the bottom left-hand corner (minimum values of X and Y) and top right-hand corner (maximum values of X and Y). These are known as (PWA_min_X,PWA_min_Y) and (PWA_max_X,PWA_max_Y). In addition, an offset co-ordinate is required to define which part of the Logical Work Area (LWA) is being displayed by the window; this co-ordinate is relative to the LWA origin, and it is known as (Scroll_X_offset, Scroll_Y_offset). It can be seen that, by adjusting the values of these offsets, it is possible to control which part of the LWA is visible within the PWA.

Note: confusingly, the Scroll_X_offset is the offset of the left edge of the PWA from the left edge of the LWA, but the Scroll_Y_offset is the offset of the *top* edge of the PWA from the *bottom* edge of the LWA. The reason for this appears to be that graphics origins are usually at the bottom left corner of the display, whereas text origins are usually at the top left.

Finally, the LWA origin may be displaced from the graphics origin of the display, thus adding another level of complexity to co-ordinate calculations! The co-ordinate of the LWA origin (and bottom left-hand corner) is defined by (LWA_min_X,LWA_min_Y) and its other extreme by the co-ordinate (LWA_max_X,LWA_max_Y).

These systems of co-ordinates allow our application and the Window Manager to decide where on the display to plot text and graphics. All plotting is achieved relative to the LWA origin, so co-ordinates must be calculated in these terms before plotting. Furthermore, the Window Manager provides the moveable parts of the scroll bars automatically, so it needs to be able to divine what amount of the LWA is visible in each direction.

Programming Using the Window Manager

Writing applications which take advantage of the Window Manager requires an approach which programmers, whose experience is limited to BASIC, may find a little unusual: window applications respond passively to 'advice' from the Window Manager about what to do next, rather than actively testing for the occurrence of particular conditions. This approach, known as 'event-driven' programming, allows the Window Manager to pass information to each window application selectively, in much the same way as the BBC MOS passes information to Sideways ROMs. This is done by

using a system of 'reason codes'. Such selectivity allows the Window Manager to time-slice the computer's processing power and thus provide a limited form of multi-tasking to window applications. Future versions of the OS are expected to offer true multi-tasking by extending this system, hopefully without requiring much re-writing of window applications.

Before we move on to examine the stages involved in getting an application running under the Window Manager, it is worth noting that most of the work involved needs to be completed before testing begins! Do not expect to be able to test a window application stage by stage, since there is too much interaction between the components for this to be feasible. Instead, it is better to use the examples herein as a template for developing your own applications and modify them, rather than start from scratch.

In this section we shall make extensive reference to the 'WimpMaze' example – listing 13.1 at the end of the chapter. This is a window application which makes use of many, though not all, of the facilities provided by the Window Manager. It randomly generates a rectangular maze within a window and allows the user to traverse it, using the mouse to leave a trail of dots whenever the left-hand button is pressed. These dots may be deleted by back-tracking over their trail while holding down the same button. The entrance to the maze is always in the bottom left corner and the exit in the top right corner. The scroll bars may be used to change the portion of the maze which is visible at any moment. You might like to try running the application to try it out before reading on.

Writing a Window Manager Application
The Stages

To get a window application started, the first thing to do is to call the Window Manager's initialisation SWI "Wimp_Initialise". This takes no parameters and simply returns the version number of the Window Manager software being used. Having done this we need to set up a control block (block%) containing the various parameters which define the window. These parameters are arranged within the control block as show on the page opposite:

Offset	Parameter (decimal)
!0	PWA_min_X
!4	PWA_min_Y
!8	PWA_max_X
!12	PWA_max_Y
!16	Scroll_X_offset initial value
!20	Scroll_Y_offset initial value
!24	Handle for window to overlay (–1 means top)
!28	Window control flags (see below)
?32	Title bar foreground colour
?33	Title bar background colour
?34	PWA foreground colour
?35	PWA background colour
?36	Scroll bar foreground colour
?37	Scroll bar background colour
?38	Title bar highlight colour
?39	Reserved
!40	LWA_min_X
!44	LWA_min_Y
!48	LWA_max_X
!52	LWA_max_Y
!56	Icon presence flags
!60	Reserved
!64	Reserved
!68	Reserved
!72	Window title string, terminated by ASCII 13 (up to 11 characters)
!84	Total icons in this window
!88	Icon data hereafter

Once these parameters are set up we need to call the Window Manager to tell it to create the window (FNcreatemain). Note that this call does not actually cause the window to appear; it merely copies the block parameters into the Window Manager's workspace and returns a handle for the window (ourwindow%).

A number of flag bits are included at offset !28 in the block: they have the following effects if set on entry:

Bit	Meaning when set
0	Title bar present
1	Window may be moved
2	Vertical scroll bar present
3	Horizontal scroll bar present

Bit	Meaning when set
4	Window only contains icons
5	Window is a sub-window of another
6	Window may be moved off display
7	Window has no 'back' or 'close' buttons

The following bits are flags set by the Window Manager to return status information and have no effect if set by the application:

16	The window is open
17	The window is un-obscured
18	The window is full size

To make the window appear we put the window's handle at the start of a block (the remaining space in the block is reserved for parameters to be returned) and call the SWI "Wimp_OpenWindow" (PROCopenwindow). We now have an empty window on the display.

The Polling Loop

Because window applications are event-driven, the main focus of any application is the 'polling loop' in which the application repeatedly asks the Window Manager what to do next. The Window Manager provides the SWI "Wimp_Poll" to achieve this: it takes a function 'mask' in R0 (see below) and a pointer to a block (for results returned) in R1. The call returns a reason code in R0 (reason%) and a pointer to the result block in R1 (info%). The reason code indicates the action to be taken by the application from the following list:

Reason Code	Meaning
0	No activity required
1	Window needs re-drawing
2	Window needs to be (re)opened *
3	Window needs to be closed *
4	Pointer leaving window
5	Pointer entering window
6	Mouse buttons have changed
7	User 'dragging' window *
8	User pressed a key
9	User selected a menu
10	User 'scrolling' window *

When calling "Wimp_Poll" the bits in the function mask word supplied in R0 can be used to disable many of the reason codes listed above – for a

given reason code number, the bit with the same number should be set to disable it (those marked with '*' may not be disabled). The normal requirement is for all events to be enabled, in which case a mask of zero is used.

Dealing with Reason Codes

For an application to work correctly it needs to respond to many of these reason codes by taking appropriate action, frequently by calling other Window Manager routines. Each reason code indicates some effect initiated by the user's actions and they are detailed individually in the following pages.

0 No Activity

The zero reason code indicates that the user has not taken any actions which affect the application. Usually this reason code would be ignored, but certain applications will need to update the display from time to time (for example, clocks) and this 'null' reason code provides an opportunity to perform this kind of action.

In listing 13.1 the null reason code triggers PROCdothatthing which checks to see if the user has quit the program or the end of the maze has been reached. If not, it checks the state of the mouse buttons and updates the display accordingly. The correct mechanism for this kind of mouse sensing is to use reason code six, but because that only indicates a change of state of the buttons rather than their current state our lazy programmer has opted for this (slightly naughty) approach!

1 Window Needs Re-drawing

This reason code is supplied to indicate the start of the sequence required to update a window on the display, perhaps because it has been moved, re-sized or become visible because of changes to the windows formerly on top of it. Of course, this process is also required the very first time a window is displayed.

The parameter block, pointed to by R1, simply contains the handle of the window concerned at offset R1+0.

To respond to the re-draw request, the application must first ask the Window Manager to re-draw the parts it is responsible for. This is achieved by calling the SWI "Wimp_RedrawWindow" with a parameter block (just containing the window handle) pointed to by R1 (PROCredraw).

The result of this call is a flag in R0 which indicates whether there are more sections to re-draw.

At this point we must divert for a moment to consider the re-drawing process. If you imagine a number of windows overlapping on the display then, when the rearmost is moved to the front, the minimum area that needs to be redrawn (to fully update the display) consists of those rectangles of the window which have just become visible.

The Window Manager provides the SWI "Wimp_GetRectangle" which should be called repeatedly and will return a flag in R0, indicating if there are any more rectangles to be re-drawn, and a parameter block, pointed to by R1, containing the co-ordinates of any such rectangles, to allow the application to divine what area it needs to re-draw.

The format of the parameter block returned by "Wimp_GetRectangle" is as follows:

Offset	Parameter
!0	Window handle
!4	PWA_min_X
!8	PWA_min_Y
!12	PWA_max_X
!16	PWA_max_Y
!20	Scroll_X_offset
!24	Scroll_Y_offset
!28	Graphics (clip) window min_X
!32	Graphics (clip) window min_Y
!36	Graphics (clip) window max_X
!40	Graphics (clip) window max_Y

Note: this information is in the same format as is returned by the SWIs "Wimp_RedrawWindow" and "Wimp_UpdateWindow" (see later) so as to remove the need to modify any parameters before passing them on.

Thus the application needs to repeatedly examine the flag in R0 and, while the flag is set, repeatedly extract and convert the PWA parameters from the block to allow it to re-draw the appropriate areas. In practice, many lazy programmers will be tempted to rely on the fact that the graphics clip window is set to the correct values and thus re-draw more than is necessary. This is an effective but slow approach and is rather against the spirit of efficiency which the rectangle sectioning system tries to promote – please resist the temptation!

To summarise then, the whole process might be 'pseudo-coded' as follows:

```
Reason code 1 received
    SYS "Wimp_RedrawWindow" ,,block% TO flag%
        WHILE flag%
            extract co-ordinates
            re-draw the specified rectangle
            SWI "Wimp_GetRectangle" ,,block% TO flag%
        ENDWHILE
```

You can see exactly this kind of code in PROCredraw in listing 13.1. Because the maze cells are of fixed size, the program relates the re-draw rectangle to the cells of the maze and then plots them using PROCdrawcell.

2 Window Needs to be (Re)Opened

This reason code is produced whenever a window is brought to the front, re-sized, or scrolled. Normally it is enough simply to pass on the parameter block returned on this reason code to the SWI "Wimp_OpenWindow". Some applications may wish to set flags to indicate that a particular kind of window has been opened.

In listing 13.1 we simply pass on the parameters to "Wimp_OpenWindow" untouched.

The parameters returned by this call are in the block pointed to by R1 at the following offsets:

Offset	Parameter
!0	Window handle
!4	New PWA_min_X
!8	New PWA_min_Y
!12	New PWA_max_X
!16	New PWA_max_Y
!20	New Scroll_X_offset
!24	New Scroll_Y_offset
!28	Window handle of window to put on top

Note: the format of this parameter block is the same as that required by SWI "Wimp_OpenWindow" so it may be passed on unaltered.

3 Window Needs to be Closed

This reason code has the opposite meaning to the one above – it indicates that the user has clicked on the 'close' box of the window. Once again, it is almost sufficient to pass on the parameter (the window handle, at R1+0) to the SWI "Wimp_CloseWindow". Usually though, your application will also

want to set a flag indicating that the window has been closed and that the application should therefore shut down any data relating to the window. This is precisely what happens in the example program, where the flag 'exitwhenredrawn%' is set to indicate impending termination.

4 Pointer Leaving Window

This reason code is generated each time the mouse pointer is moved out of the PWA, the trigger point being the edge of the PWA and not the edge of the whole window. The window handle is returned at R1+0 in the usual way.

The normal use of this is to prevent depression of the mouse buttons having any effect while the pointer is outside the window, or to close menus and other 'pop-up' devices automatically as the pointer moves away from them.

The example program sets the flag 'ourwindow%' (which contains the handle of our window) to –1 on this call so as to prevent mouse button activity elsewhere. It is reset to the window handle by the next reason code.

5 Pointer Entering Window

This reason code is complementary to reason code 4 – it is generated each time the pointer enters the PWA part of the window. The window handle is returned at R1+0 in the usual way. This call allows for windows which appear automatically when the pointer enters a particular part of the display.

Listing 13.1 uses this call to make the flag 'ourwindow%' valid again by setting it to the window handle at R1+0. This flag is used to determine whether mouse buttons have an effect elsewhere in the program.

6 Mouse Buttons have Changed

This reason code is issued each time the mouse buttons change state. It does not indicate which mouse buttons are actually being held down; only that they have been pressed or released. The following parameters are returned in the parameter block for this reason code:

Offset	Parameter
!0	Mouse X co-ordinate
!4	Mouse Y co-ordinate
!8	New state of mouse buttons
!12	Window handle (or –1 if outside all windows)
!16	Icon handle (or –1 if not on an icon)
!20	Old state of mouse buttons

The format of the mouse state flags at !8 are as follows:

Bit	Meaning
0	Right button pressed (adjust)
1	Middle button pressed (menu)
2	Left button pressed (select)
4	Dragging with right button
6	Dragging with left button
8	Single click of right button
10	Single click of left button

Listing 13.1 uses this call to identify when the middle (menu) button has been pressed within its window causing the pop-up menu to appear (PROCmouse, PROCsetupmenu).

7 User Dragging Window

This reason code is returned at the end of a drag sequence whose effect is application-specific. Consult the Programmer's Reference Manual for more information.

8 User Pressed a Key

This reason code is issued if the user presses a key when the pointer is within one of the application's windows. The Window Manager attempts to display text at the position of the text cursor (known as the 'caret') whose position may be set with the pointer. This reason code is accompanied by the following parameter block:

Offset	Parameter
!0	Window handle in which caret appears
!4	Icon handle (or –1 if no icon)
!8	Caret_X_offset (within PWA)
!12	Caret_Y_offset (within PWA)
!16	Height of caret
!20	Position of carat within text string
!24	Code of key pressed

The entry of characters into windows is beyond the scope of this book and the reader is advised to consult the Programmer's Reference Guide for more information.

9 User Selected a Menu

This reason code is returned when the user has made a selection from an active menu (which has been set up using SWI "Wimp_CreateMenu"). The parameter block contains the item numbers of the menu items selected, with R1+0 containing the item from the first menu level, R1+4 containing the item number from the second menu level and so on up to a terminating –1. See the section on menus later for more information.

Closing Down the Application Window

Having established the mechanisms for getting a window onto the screen and running an application behind it, we need to be able to shut down an application cleanly. This is achieved with the assistance of two SWIs – "Wimp_DeleteWindow" and "Wimp_CloseDown".

"Wimp_DeleteWindow" takes a window handle at R1+0 and removes that window from the display. "Wimp_CloseDown" shuts down the entire Window Manager and clears the display.

Window Manager Support for Menus

Besides providing overall control of windows, the Window Manager also contains a number of routines which offload the more mundane aspects of dealing with user selections from 'menus'. The idea behind menus, as we saw earlier, is to minimise the scope for user error by predefining the valid selections from the menu. The OS Window Manager supports multi-level or 'hierarchical' menus, which allow complex sequences of selections to be made from almost any number of menu levels. The user causes a menu to appear by clicking on the middle mouse button: the application determines which window the pointer is over at the time and the Window Manager displays the appropriate menu and allows the user to make a selection. By taking advantage of the Window Manager support for menus, application programmers may be relieved of the burden of dealing with menu selections and simply concern themselves with the effects of these selctions. The results of selections are returned in a control block provided by "Wimp_Poll" reason code nine.

The Structure of Menus

A menu is described by the application to the Window Manager in a similar way to that used for windows. In fact, menus are really just windows whose contents are predefined – this predefinition allowing the Window Manager to do most of the work in dealing with user selections.

A menu consists of a list of entries displayed as a column of lines of text within a window. The window has none of the control buttons and scroll bars associated with normal windows, but instead each of the menu entries may have three graphical effects associated with it:

The first is that it may be 'dimmed', a shading effect which indicates that it is not valid to select the entry.

The second is a 'tick' or check mark which indicates that the entry is already selected or is a default value.

The third is an 'arrow' facing off the right-hand side of an entry which indicates the presence of a sub-menu or window. Moving the mouse to the right over one of these arrows causes the sub-menu to appear to the right of the first menu, and this process may continue through more levels of sub-menu *ad nauseam*. The Window Manager attempts to be intelligent about the positioning of menus so, if the sub-menu is selected from a menu which is already near the right-hand edge of the screen it will be displaced left to ensure it is completely visible.

Programming Menus

To prepare the Window Manager for the handling of menus it is necessary to call the SWI "Wimp_CreateMenu" once for each menu in the application. This call takes a pointer to the menu's control block and a co-ordinate pair which specifies where it should initially appear on the display. The program should then return to the main polling loop to await the issue of reason code nine which indicates that a selection has actually been made.

SWI Wimp_CreateMenu (SWI &400D4)
Create a New Menu Structure

This call advises the Window Manager that a menu structure should be installed in its tables. On entry, R0 must either contain –1 (in which case all menus are closed) or a pointer to the menu definition block for the new

menu. In the latter case (R2,R3) give the co-ordinates of the top left-hand corner of the menu when it is displayed.

The menu control block consists of a series of entries, each of which is in the same format. The first defines the top level menu and its entries, with pointers leading the way to the sub-menu definitions and so on. The structure of the menu control block is as follows:

Offset	Contents
!0	Menu title string terminated by zero
?12	Menu title foreground colour
?13	Menu title background colour
?14	Work area foreground colour
?15	Work area background colour
!16	Width of sub-menus (pixels)
!20	Height of sub-menus (pixels)
!24	Separation between menu entries (pixels)
!28	Menu entries start here

The menu entries consist of 24-byte blocks which contain the names and flags for each entry. Their format is as follows:

Offset	Contents
!0	Flags: Bit 0 – Set means 'tick' the entry Bit 1 – Set means follow this entry with a dotted divider Bit 2 – Set means this entry may be overwritten by user Bit 7 – Set means this is the last item in the menu
!4	Pointer to sub-menu or handle of sub-window or –1 (for neither)
!8	Icon flags
!12	Text of entry terminated by zero (11 characters maximum)

Where an entry has a sub-menu, the presence of a valid pointer or window handle at offset !4 causes the arrow to appear automatically. For sub-menus, the offset of the sub-menu in the menu control block should be inserted here; for sub-windows, the window handle of the appropriate window should be inserted.

The structure of sub-menu entries is just the same as that for the first menu, with all the information indicated above repeated as necessary.

Menus in the WimpMaze Example

The example program at the end of this chapter makes limited use of the menu facility by providing a menu which permits the foreground and background colours of the display to be changed. This is set up when the middle button is pressed (PROCmouse) by the procedure PROCsetupmenu. Subsequently, the Wimp_Poll reason code is issued and PROChandlemenu is called to extract the parameters returned by the reason code and change the display colours accordingly.

Note: do not be discouraged by the length of PROCsetupmenu. It is rather long-winded since each menu entry has to be individually defined, but the menu system itself is not at all complicated. When setting up menus, you may find it easiest to type a 'dummy' entry and duplicate it several times (eg, by using the copy facility in the Arm BASIC Editor).

The WIMP routines described previously are sufficient to carry out most window based operations. The WIMP Manager provides several more advanced calls to help in manipulating windows in specialised ways. A complete list of the facilities provided is given below.

SWI Number	Routine Name	SWI Number	Routine Name
&400C0	Wimp_Initialise	&400C1	Wimp_CreateWindow
&400C2	Wimp_CreateIcon	&400C3	Wimp_DeleteWindow
&400C4	Wimp_DeleteIcon	&400C5	Wimp_OpenWindow
&400C6	Wimp_CloseWindow	&400C7	Wimp_Poll
&400C8	Wimp_RedrawWindow	&400C9	Wimp_UpdateWindow
&400CA	Wimp_GetRectangle	&400CB	Wimp_GetWindowState
&400CC	Wimp_GetWindowInfo	&400CD	Wimp_SetIconState
&400CE	Wimp_GetIconState	&400CF	Wimp_GetPointerInfo
&400D0	Wimp_DragBox	&400D1	Wimp_ForceRedraw
&400D2	Wimp_SetCaretPosition	&400D3	Wimp_GetCaretPosition
&400D4	Wimp_CreateMenu	&400D5	Wimp_DecodeMenu
&400D6	Wimp_WhichIcon	&400D7	Wimp_SetExtent
&400D8	Wimp_SetPointerShape	&400D9	Wimp_OpenTemplate
&400DA	Wimp_CloseTemplate	&400DB	Wimp_LoadTemplate
&400DC	Wimp_ProcessKey	&400DD	Wimp_CloseDown

To assist in the design and creation of WIMP windows, Acorn has produced a window designer. This allows the various parameters defining a window to be easily selected and varied. A window definition file, called a template, is then produced by the designer. The WIMP manager provides two SWI calls to allow this template file to be loaded into the WIMP system. The window

definitions held within it are then available to the WIMP as if they had been created using WIMP_CreateWindow.

```
 10 REM >List13/1
 20 REM by Nicholas van Someren
 30 REM Archimedes OS: A Dabhand Guide
 40 REM (c) Copyright AvS and NvS 1988
 50 :
 60 ON ERROR PROCerror:END
 70 :
 80 REM Select mode 12 and set up the palette.
 90 :
100 MODE 12
110 VDU 19,15,16,16,16,160
120 backcol%=3
130 forcol%=4
140 VDU 19,8,backcol%
150 VDU 19,9,forcol%
160 :
170 REM Initialise the Wimp.
180 :
190 SYS "Wimp_Initialise" TO version%
200 :
210 REM Set up maze variables.
220 :
230 side%=20
240 area%=side%*side%
250 DIM a% area%,b% area%*2,block% 255
260 :
270 REM Reserve space for menus and enter
280 REM the main loop.
290 :
300 DIM menu% 1023
310 :
320 REPEAT
330 :
340 REM Create a new maze. Each cell will be
350 REM 100 by 100 pixels on the screen.
360 :
370 PROCnewmaze
380 scale%=100
390 :
400 REM We'll start in the bottom left corner.
410 weareX%=0
420 weareY%=0
430 ?a%=?a% OR 16
440 :
450 REM Define and open our window.
460 :
470 ourwindow%=FNcreatemain
480 PROCopenwindow(ourwindow%)
490 :
500 REM Set up flags to indicate if user has solved
```

```
510 REM maze or left the program.
520 :
530 solved%=0
540 solvedwhendrawn%=0
550 exit%=0
560 exitwhendrawn%=0
570 :
580 REM The polling loop - polls the Wimp until
590 REM maze is solved or game is quit.
600 :
610 REPEAT
620 SYS "Wimp_Poll",,block% TO reason%,info%
630 CASE reason% OF
640 WHEN 0:PROCdothatthing :REM Null reason code
650 WHEN 1:PROCredraw(info%)
660 WHEN 2:SYS "Wimp_OpenWindow",,info%
670 WHEN 3:SYS "Wimp_CloseWindow",,info%:exitwhendrawn%=TRUE
680 WHEN 4:ourwindow%=-1           :REM Pointer leaving
690 WHEN 5:ourwindow%=info%!0      :REM Pointer entering
700 WHEN 6:PROCmouse(info%)        :REM Mouse buttons
710 WHEN 7                         :REM User Drag
720 WHEN 8                         :REM Key Pressed
730 WHEN 9:PROChandlemenu(info%)   :REM Menu Select
740 WHEN 10    :REM Scroll Request
750 OTHERWISE
760 ENDCASE
770 UNTIL solved% OR exit%
780 :
790 REM Delete the old window and repeat if solved.
800 REM Otherwise, exit tidily.
810 :
820 PROCdeletewindow(handle%)
830 UNTIL exit%
840 PROCfinish
850 END
860 :
870 DEF FNcreatemain
880 block%!0=200   :REM Window, min X
890 block%!4=200   :REM Window, min Y
900 block%!8=1000  :REM Window, max X
910 block%!12=700  :REM Window, max Y
920 block%!16=-20  :REM X scroll offset
930 block%!20=480  :REM Y scroll offset
940 block%!24=-1   :REM Open at the top
950 block%!28=%1111    :REM Flags
960 block%?32=7    :REM Title fore' colour
970 block%?33=0    :REM Title back' colour
980 block%?34=9    :REM Work fore' colour
990 block%?35=8    :REM Work back' colour
1000 block%?36=5   :REM Scroll outer colour
1010 block%?37=6   :REM Scroll inner colour
1020 block%?38=1   :REM Highlighted title colour
1030 block%?39=0   :REM Reserved
```

```
1040 block%!40=-20                :REM Work area extent, min X
1050 block%!44=-20                :REM Work area extent, min Y
1060 block%!48=side%*scale%+20    :REM Work area extent, max X
1070 block%!52=side%*scale%+20    :REM Work area extent, max Y
1080 block%!56=%00000111000000000000000000011001     :REM Flags
1090 block%!60=0                  :REM Work area button type
1100 block%!64=0                  :REM Sprite control pointer
1110 block%!68=0                  :REM Reserved
1120 $(block%+72)="My Window."    :REM Window title
1130 block%!84=0                  :REM Number of icons
1140 :
1150 REM Create the window and return its handle.
1160 :
1170 SYS "Wimp_CreateWindow",,block% TO handle%
1180 =handle%
1190 :
1200 REM Open a window: set up block% with the
1210 REM handle, read information about the
1220 REM window and use this information to open it.
1230 :
1240 DEF PROCopenwindow(handle%)
1250 block%!0=handle%
1260 SYS "Wimp_GetWindowState",,block%
1270 SYS "Wimp_OpenWindow",,block%
1280 ENDPROC
1290 :
1300 REM Close a window.
1310 :
1320 DEF PROCdeletewindow(handle%)
1330 block%!0=handle%
1340 SYS "Wimp_DeleteWindow",,block%
1350 ENDPROC
1360 :
1370 REM Keep redrawing the maze until there is
1380 REM nothing else to redraw.
1390 :
1400 DEF PROCredraw(block%)
1410 SYS "Wimp_RedrawWindow",,block% TO flag%
1420 WHILE flag%
1430 :
1440 REM Calculate the edges of the graphics window.
1450 :
1460 rectminX=block%!28
1470 rectminY=block%!32
1480 rectmaxX=block%!36
1490 rectmaxY=block%!40
1500 :
1510 REM Calculate the size of the graphics window.
1520 :
1530 sizeX=rectmaxX-rectminX
1540 sizeY=rectmaxY-rectminY
1550 :
1560 REM Calculate window position on PWA.
```

```
1570 :
1580 vertminX=block%!20+block%!28-block%!4
1590 vertmaxY=block%!24+block%!40-block%!16
1600 vertmaxX=vertminX+sizeX
1610 vertminY=vertmaxY-sizeY
1620 :
1630 REM Calculate window position on work area.
1640 :
1650 offX=rectminX-vertminX
1660 offY=rectminY-vertminY
1670 :
1680 REM Redraw each maze cell within the window.
1690 :
1700 FOR I%=(vertminX DIV scale%) TO (vertmaxX DIV scale%)
1710 FOR J%=(vertminY DIV scale%) TO (vertmaxY DIV scale%)
1720 PROCdrawcell(I%,J%)
1730 NEXT
1740 NEXT
1750 :
1760 REM Check if there is anything else to draw.
1770 :
1780 SYS "Wimp_GetRectangle",,block% TO flag%
1790 ENDWHILE
1800 ENDPROC
1810 :
1820 REM Draw a maze cell if co-ordinates are within maze.
1830 :
1840 DEF PROCdrawcell(X%,Y%)
1850 LOCAL C%
1860 IF X%>=0 AND X%<side% AND Y%>=0 AND Y%<side% THEN
1870 C%=?(a%+X%+side%*Y%)
1880 MOVE X%*scale%+offX,Y%*scale%+offY
1890 :
1900 REM Draw left, top, right and bottom sides.
1910 :
1920 PLOT 1+((C% AND 1)=1),0,(scale%-1)
1930 PLOT 1+((C% AND 8)=8),(scale%-1),0
1940 PLOT 1+((C% AND 4)=4),0,-(scale%-1)
1950 PLOT 1+((C% AND 2)=2),-(scale%-1),0
1960 :
1970 REM Move to the centre and draw the circle.
1980 :
1990 MOVE (X%+.5)*scale%+offX,(Y%+.5)*scale%+offY
2000 IF (C% AND 16)=16 THEN
2010 PLOT 153,scale%/3,0
2020 ELSE
2030 PLOT 155,scale%/3,0
2040 ENDIF
2050 ENDIF
2060 ENDPROC
2070 :
2080 DEF PROCmouse(block%)
2090 IF (block%?8 AND 2)=2 AND (block%!12)=handle% THEN
```

```
2100  PROCsetupmenu
2110  SYS "Wimp_CreateMenu",,menu%,block%!0,block%!4
2120  ENDIF
2130  ENDPROC
2140  :
2150  REM Set up colour sub-menu.
2160  :
2170  DEF PROCsetupmenu
2180  $menu%="Colours"+CHR$0
2190  menu%?12=4                              :REM Colours
2200  menu%?13=3
2210  menu%?14=6
2220  menu%?15=2
2230  menu%!16=200                            :REM Width
2240  menu%!20=32                             :REM Height of entries
2250  menu%!24=16                             :REM Entry gap height
2260  menu%!28=0                              :REM No flags
2270  menu%!32=menu%+80                       :REM Pointer to sub menu
2280  menu%!36=%00000000000000000000111001 :REM Menu flags
2290  menu%?39=forcol%<<4 OR (forcol% EOR %111)  :REM Icon cols
2300  $(menu%+40)="Foreground"+CHR$0          :REM Entry text
2310  menu%!52=&80                            :REM Flag as last entry
2320  menu%!56=menu%+300
2330  menu%!60=%00000000000000000000111001
2340  menu%?63=backcol%<<4 OR (backcol% EOR %111)
2350  $(menu%+64)="Background"+CHR$0
2360  menu%!80=0                              :REM Header (as above)
2370  menu%?92=4
2380  menu%?93=3
2390  menu%?94=6
2400  menu%?95=15
2410  menu%!96=128
2420  menu%!100=40
2430  menu%!104=0
2440  mb%=menu%+108
2450  mb%!0=0
2460  IF forcol%=%000 THEN mb%?0=1
2470  mb%!4=-1                                :REM No sub-menu
2480  mb%!8=%00000111000000000000000000111001    :REM Flags
2490  IF backcol%=%000 THEN mb%!8=mb%!8 OR 1<<22
2500  $(mb%+12)="Black"                       :REM Entry text
2510  mb%+=24                                 :REM Set mb% - next entry
2520  mb%!0=0
2530  IF forcol%=%001 THEN mb%?0=1
2540  mb%!4=-1
2550  mb%!8=%00010110000000000000000000111001
2560  IF backcol%=%001 THEN mb%!8=mb%!8 OR 1<<22
2570  $(mb%+12)="Red"
2580  mb%+=24
2590  mb%!0=0
2600  IF forcol%=%010 THEN mb%?0=1
2610  mb%!4=-1
2620  mb%!8=%00100101000000000000000000111001
```

```
2630  IF backcol%=%010 THEN mb%!8=mb%!8 OR 1<<22
2640  $(mb%+12)="Green"
2650  mb%+=24
2660  mb%!0=0
2670  IF forcol%=%011 THEN mb%?0=1
2680  mb%!4=-1
2690  mb%!8=%00110100000000000000000000111001
2700  IF backcol%=%011 THEN mb%!8=mb%!8 OR 1<<22
2710  $(mb%+12)="Yellow"
2720  mb%+=24
2730  mb%!0=0
2740  IF forcol%=%100 THEN mb%?0=1
2750  mb%!4=-1
2760  mb%!8=%01000011000000000000000000111001
2770  IF backcol%=%100 THEN mb%!8=mb%!8 OR 1<<22
2780  $(mb%+12)="Blue"
2790  mb%+=24
2800  mb%!0=0
2810  IF forcol%=%101 THEN mb%?0=1
2820  mb%!4=-1
2830  mb%!8=%01010010000000000000000000111001
2840  IF backcol%=%101 THEN mb%!8=mb%!8 OR 1<<22
2850  $(mb%+12)="Magenta"
2860  mb%+=24
2870  mb%!0=0
2880  IF forcol%=%110 THEN mb%?0=1
2890  mb%!4=-1
2900  mb%!8=%01100001000000000000000000111001
2910  IF backcol%=%110 THEN mb%!8=mb%!8 OR 1<<22
2920  $(mb%+12)="Cyan"
2930  mb%+=24
2940  mb%!0=&80
2950  IF forcol%=%111 THEN mb%?0=&81
2960  mb%!4=-1
2970  mb%!8=%01110000000000000000000000111001
2980  IF backcol%=%111 THEN mb%!8=mb%!8 OR 1<<22
2990  $(mb%+12)="White"
3000  mb%+=24
3010  menu%!300=0                  :REM Sub-menu for background
3020  menu%?312=4
3030  menu%?313=3
3040  menu%?314=6
3050  menu%?315=15
3060  menu%!316=128
3070  menu%!320=40
3080  menu%!324=0
3090  mb%=menu%+328
3100  mb%!0=0                      :REM All entries as before
3110  IF backcol%=%000 THEN mb%?0=1
3120  mb%!4=-1
3130  mb%!8=%00000111000000000000000000111001
3140  IF forcol%=%000 THEN mb%!8=mb%!8 OR 1<<22
3150  $(mb%+12)="Black"
```

```
3160  mb%+=24
3170  mb%!0=0
3180  IF backcol%=%001 THEN mb%?0=1
3190  mb%!4=-1
3200  mb%!8=%00010110000000000000000000111001
3210  IF forcol%=%001 THEN mb%!8=mb%!8 OR 1<<22
3220  $(mb%+12)="Red"
3230  mb%+=24
3240  mb%!0=0
3250  IF backcol%=%010 THEN mb%?0=1
3260  mb%!4=-1
3270  mb%!8=%00100101000000000000000000111001
3280  IF forcol%=%010 THEN mb%!8=mb%!8 OR 1<<22
3290  $(mb%+12)="Green"
3300  mb%+=24
3310  mb%!0=0
3320  IF backcol%=%011 THEN mb%?0=1
3330  mb%!4=-1
3340  mb%!8=%00110100000000000000000000111001
3350  IF forcol%=%011 THEN mb%!8=mb%!8 OR 1<<22
3360  $(mb%+12)="Yellow"
3370  mb%+=24
3380  mb%!0=0
3390  IF backcol%=%100 THEN mb%?0=1
3400  mb%!4=-1
3410  mb%!8=%01000011000000000000000000111001
3420  IF forcol%=%100 THEN mb%!8=mb%!8 OR 1<<22
3430  $(mb%+12)="Blue"
3440  mb%+=24
3450  mb%!0=0
3460  IF backcol%=%101 THEN mb%?0=1
3470  mb%!4=-1
3480  mb%!8=%01010010000000000000000000111001
3490  IF forcol%=%101 THEN mb%!8=mb%!8 OR 1<<22
3500  $(mb%+12)="Magenta"
3510  mb%+=24
3520  mb%!0=0
3530  IF backcol%=%110 THEN mb%?0=1
3540  mb%!4=-1
3550  mb%!8=%01100001000000000000000000111001
3560  IF forcol%=%110 THEN mb%!8=mb%!8 OR 1<<22
3570  $(mb%+12)="Cyan"
3580  mb%+=24
3590  mb%!0=&80
3600  IF backcol%=%111 THEN mb%?0=&81
3610  mb%!4=-1
3620  mb%!8=%01110000000000000000000000111001
3630  IF forcol%=%111 THEN mb%!8=mb%!8 OR 1<<22
3640  $(mb%+12)="White"
3650  mb%+=24
3660  ENDPROC
3670  :
3680  DEF PROChandlemenu(block%)
```

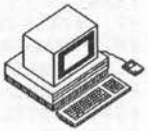
```
3690  IF block%!0<>-1 AND block%!4<>-1 THEN
3700  VDU 19,9-(block%!0),block%!4,0,0,0
3710  IF block%!0=1 backcol%=block%!4 ELSE forcol%=block%!4
3720  ENDIF
3730  ENDPROC
3740  :
3750  REM PROCdothatthing is called when the Wimp is idle.
3760  :
3770  DEF PROCdothatthing
3780  IF solvedwhendrawn% OR exitwhendrawn% THEN
3790  IF solvedwhendrawn% solved%=TRUE ELSE exit%=TRUE
3800  ELSE
3810  IF weareX%=side%-1 AND weareY%=side%-1 THEN
3820  :
3830  REM The maze has been solved so set appropriate
3840  REM flags, delete window etc.
3850  :
3860  solvedwhendrawn%=TRUE
3870  block%!0=handle%
3880  PROCdeletewindow(ourwindow%)
3890  scale%=(scale% DIV 5) AND &FFC
3900  ourwindow%=FNcreatemain
3910  PROCopenwindow(ourwindow%)
3920  ELSE
3930  :
3940  REM Check pointer, check if a button is pressed
3950  REM and pointer is inside window.
3960  :
3970  MOUSE areatX%,areatY%,buttons
3980  IF ourwindow%=handle% AND buttons<>0 THEN
3990  :
4000  REM Examine window state, determine the
4010  REM relationship between window and work area
4020  REM and find which maze cell the pointer is in.
4030  :
4040  block%!0=handle%
4050  SYS "Wimp_GetWindowState",,block%
4060  offX=block%!4-block%!20
4070  offY=block%!16-block%!24
4080  areatX%=(areatX%-offX)DIV scale%
4090  areatY%=(areatY%-offY)DIV scale%
4100  IF (weareX%<>areatX% OR weareY%<>areatY%) AND
areatX%>=0 AND areatX%<side% AND areatY%>=0 AND areatY%<side% THEN
4110  :REM If the mouse can move the new cell, do so
4120  IF weareX%=areatX%+1 AND weareY%=areatY% AND (?(a%+weareX%
+side%*weareY%)AND1) PROCmousemove
4130  IF weareX%=areatX% AND weareY%=areatY%+1 AND (?(a%+weareX%
+side%*weareY%)AND2) PROCmousemove
4140  IF weareX%=areatX%-1 AND weareY%=areatY% AND (?(a%+weareX%
+side%*weareY%)AND4)=4 PROCmousemove
4150  IF weareX%=areatX% AND weareY%=areatY%-1 AND  (?(a%+ weareX%
+side%*weareY%)AND8)=8 PROCmousemove
4160  ENDIF
```

```
4170  ENDIF
4180  ENDIF
4190  ENDIF
4200  ENDPROC
4210  :
4220  REM Move the pointer when possible.
4230  :
4240  DEF PROCmousemove
4250  IF (?(a%+areatX%+side%*areatY%)AND 16)=16 THEN
4260  :
4270  REM Retrace step and update screen.
4280  :
4290  ?(a%+weareX%+side%*weareY%)=?(a%+weareX%+side%*weareY%) AND
&EF
4300  SYS "Wimp_ForceRedraw",handle%,weareX%*scale%+4,
weareY%*scale%+4, (weareX%+1) weale%-4,(weareY%+1)*scale%-4
4310  ELSE
4320  :
4330  REM Move to a new cell and update screen.
4340  :
4350  ?(a%+areatX%+side%*areatY%)=?(a%+areatX%+side%*areatY%) OR
&10
4360  SYS "Wimp_ForceRedraw",handle%,areatX%*scale%+4,
areatY%*scale%+4, (areatX%+1)*scale%-4,(areatY%+1)*scale%-4
4370  ENDIF
4380  weareX%=areatX%
4390  weareY%=areatY%
4400  ENDPROC
4410  :
4420  REM The maze generator.
4430  :
4440  DEF PROCnewmaze
4450  LOCAL P%,X%,Y%,L%
4460  FORI%=0TOarea%-1STEP4:a%!I%=0:NEXT
4470  P%=0
4480  PROCnewcell(RND(side%)-1,RND(side%)-1)
4490  REPEAT
4500  IF P%>3 AND RND(100)<>1 L%=RND(3)+P%-4 ELSE L%=RND(P%)-1
4510  X%=?(b%+L%):Y%=?(b%+L%+area%)
4520  P%=P%-1
4530  ?(b%+L%)=?(b%+P%):?(b%+L%+area%)=?(b%+P%+area%)
4540  Flag%=0
4550  REPEAT
4560  CASE RND(4) OF
4570  WHEN 1:IF X%<>0 IF (?(a%+X%-1+Y%*side%) AND &80)<>0
?(a%+X%+Y%*side%)=?(a%+X%+Y%*side% OR 1:?(a%+X%-1+Y% *side%)
=?(a%+X%-1+Y%*side%) OR 4:Flag%=-1
4580  WHEN 2:IF Y%<>0 IF (?(a%+X%+(Y%-1)*side%) AND &80)<>0
?(a%+X%+Y%*side%)=?(a%+X%+Y%*side OR 2:?(a%+X%+(Y%-1)*side%)
=?(a%+X%+(Y%-1)*side%) OR 8:Flag%=-1
4590  WHEN 3:IF X%<>side%-1 IF (?(a%+X%+1+Y%*side%) AND &80)<>0
?(a%+X%+Y%*side%)=?(a%+X%+Y%*side%)OR 4:?(a%+X%+1+Y%*side%)
=?(a%+X%+1+Y%*side%) OR 1:Flag%=-1
```

```
4600  WHEN 4:IF Y%<>side%-1 IF (?(a%+X%+(Y%+1)*side%) AND &80)<>0
?(a%+X%+Y%*side%)=?(a%+X%+Y%*side OR 8:?(a%+X%+(Y%+1)*side%)
=?(a%+X%+(Y%+1)*side%) OR 2:Flag%=-1
4610  ENDCASE
4620  UNTIL Flag%
4630  PROCnewcell(X%,Y%)
4640  UNTIL P%=0
4650  X%=0:Y%=0:?(a%+X%+Y%*side%)=?(a%+X%+Y%*side%)OR1
4660  X%=side%-1:Y%=side%-1:?(a%+X%+Y%*side%)=?(a%+X%+Y%*side%)OR4
4670  ENDPROC
4680  :
4690  DEF PROCnewcell(X%,Y%)
4700  ?(a%+X%+Y%*side%)=?(a%+X%+Y%*side%) OR &C0
4710  IF X%>0 IF (?(a%+X%-1+Y%*side%) AND &40)=0 PROClistadd(X%-
1,Y%)
4720  IF X%<side%-1 IF (?(a%+X%+1+Y%*side%) AND &40)=0
PROClistadd(X%+1,Y%)
4730  IF Y%>0 IF (?(a%+X%+(Y%-1)*side%) AND &40)=0
PROClistadd(X%,Y%-1)
4740  IF Y%<side%-1 IF (?(a%+X%+(Y%+1)*side%) AND &40)=0
PROClistadd(X%,Y%+
4750 ENDPROC
4760  :
4770  DEF PROClistadd(I%,J%)
4780  ?(a%+I%+J%*side%)=?(a%+I%+J%*side%)OR &40
4790  ?(b%+P%)=I%
4800  ?(b%+P%+area%)=J%
4810  P%=P%+1
4820  ENDPROC
4830  :
4840  REM The error handler, which uses PROCfinish to be tidy.
4850  :
4860  DEF PROCerror
4870  PROCfinish
4880  PRINT REPORT$;" at line ";ERL
4890  ENDPROC
4900  :
4910  REM Shut down the Wimp system tidily.
4920  :
4930  DEF PROCfinish
4940  SYS "Wimp_CloseDown"
4950  *FX 4
4960  *POINTER 0
4970  VDU 4,26,20,12
4980  ENDPROC
```

Listing 13.1. WimpMaze.

14 : The Font Manager



The 'Font Manager' and its twin – the 'Font Painter' – allow characters to be displayed in a number of fonts, anti-aliased, scaled to any size, proportionally spaced and with optional position justification. The fonts are stored in two sets of files (known as the 'metric' and 'pixel' files) which define the shape of the font and contain pixel bit-maps of the font characters. By calling the Font Manager, an application may request a specified font in a given size and then display (optionally coloured) text in a chosen screen mode.

The Font Manager deals with the business of reading font definitions into memory (a process known as 'caching') and scaling the fonts to the required size as it does so. An area of memory may be set aside for the Font Manager's cache using the configuration option FontSize, and this limits the number of fonts which may be cached at any moment. If insufficient space is available in memory the Font Manager will delete some of the existing fonts to make room for fresh ones. Thus the application does not have to deal with the memory management issues.

The Font Painter takes the cached font definitions and paints them onto the display either by means of SWI calls or through the normal VDU channel for printing text. This latter technique is achieved by intercepting the VDU extension vector VDUXV.

So that the Font Manager and the Font Painter can communicate with each other effectively there is a fixed relationship between the scales each use. The Font Manager works in $^{1}/_{1000}$ths of a 'point', a typographer's measure which is roughly $^{1}/_{72}$nd of an inch. Thus the Font Manager's basic units are $^{1}/_{72000}$ths of an inch. The Font Painter, on the other hand, works with screen units (pixels) which it assumes to be $^{1}/_{180}$th of an inch, so there are nominally 90 pixels per inch in mode zero (which has half the theoretical maximum resolution).

Dealing with the Font Manager

The Font Manager handles the movement and scaling of font files which are stored on the selected filing system. The OS variable Font$Prefix

specifies where the font files are to be found. Font files are stored in sub-directories of a directory whose name is the name of the font, so a full pathname is derived by using the contents of Font$Prefix followed by the name of the font, eg, after entering:

```
*SET Font$Prefix $.Welcome.Fonts
```

the files for the font named Trinity will henceforth be found in:

$.Welcome.Fonts.Trinity

To allow the Font Manager to perform caching and de-caching automatically it is necessary to follow this procedure:

1. Request the font using SWI "Font_FindFont" by name, specifying its point size and screen resolution. A font 'handle' is returned by the Font Manager.
2. Use this handle in any subsequent requests to the Font Manager.
3. Advise the Font Manager that the font is no longer needed by calling SWI "Font_LoseFont" with the handle.

When an application requests a font, the Font Manager first checks the cache to see if the font is already available in the specified size. If so, the handle of the font is returned immediately; otherwise, the Font Manager loads the font information from the filing system (evicting other fonts from the cache if necessary) and returns a new handle to the application.

An Example

Listing 14.1 is a full listing of an example program which uses the Font Manager. Bearing in mind the sequence described above, here is a summary of how the program works.

First, the SWI "Font_CacheAddr" is issued to determine which version of the Font Manager is being used and to return the total size of the font cache and the amount already used. This call is made with R0 containing zero and returns with the version number*100 in R0, the amount of cache used in R1 and the total cache size in R2. The following line of BASIC achieves this:

```
SYS "Font_CacheAddr" TO version%,cacheused%,cachesize%
```

Next, three different combinations of font and size are requested from the Font Manager using SWI "Font_FindFont". This takes a pointer to the font name in R1, the width in points*16 in R2, the height in points*16 in R3

and, optionally, the X and Y resolutions of the display in pixels per inch in R4 and R5 (if zero is supplied then the default is used). The Font Manager returns a handle to the font in R0. The following line of Basic gets the handle in Trinity% for a 48 point square version of the supplied font "Trinity Medium":

```
SYS "Font_FindFont",,"Trinity.Medium",48*16,48*16 TO Trinity
```

The palette needs to be set up to anti-alias the fonts correctly in the selected mode (mode 12 in the example). Fonts are inherently anti-aliased using 16 colours, so in modes where this number of colours is not spare, the number of anti-aliasing levels must be reduced. This is achieved by issuing SWI "Font_SetPalette" which takes as parameters the background logical colour in R1, the foreground logical colour in R2, the foreground colour offset (see below) in R3, the physical background colour in R4 and the 'last' physical foreground colour in R5.

Consider the following line from the example listing 14.1:

```
SYS "Font_SetPalette",,0,1,6,&00000000,&F0F0F000
```

The last two figures indicate the start and end physical colours. These are in the form &BBGGRR00 where BB, GG and RR are the blue, green and red intensities respectively. So the line above will result in a colour scale from black (&00000000) to white (&F0F0F000). The parameters 0,1,6 determine which logical colours are used for this colour scale. The 0 indicates that logical colour 0 is to be used for the background colour, the 1 defines the colour scale as starting at logical colour 1 and the 6 means that 6 colours (excluding the background colour) are to be used in the scale. Hence, logical colours 0 to 7 are redefined to form an ascending black-to-white colour scale.

The next line in the example performs a similar function, setting logical colours 7 to 15 to form a scale from white to dark-green. Note that the example listing 14.1 uses the physical colour white to terminate one colour scale and begin another. By sharing colours in this way, a larger variety of shades may be squeezed into a limited logical colour range.

Getting Text on the Display

Having identified to the Font Manager those fonts which will need caching and having set up the palette for anti-aliasing, we may now proceed to actually display text.

The first stage is to identify the colours in which we wish to display the chosen font using the SWI "Font_SetFontColours". This SWI takes a font

handle in R0 (or 0 for the 'current' or most recently used font), a background logical colour in R1, a foreground logical colour in R2 and a foreground colour offset in R3 (as above). This call returns no results, but simply establishes the colours that will subsequently be used. The following line is used in the example listing:

```
SYS "Font_SetFontColours",Trinity%,7,8,7
```

The second stage is to ask the Font Manager to be prepared to print in the font we intend to use. This is achieved using the SWI "Font_SetFont" which simply takes the font handle in R0 and returns no results, eg:

```
SYS "Font_SetFont",Trinity%
```

Finally, we can actually pass to the Font Painter the text to be displayed, accompanied by a 'plot type' indicating the way in which it should appear. The SWI "Font_Paint" achieves this, taking a pointer to the string to be displayed in R1, the plot type in R2 (see below) and the X and Y co-ordinates where the text is to start in R3 and R4 respectively.

It is important to note that the pair of co-ordinates specifying the position where the text will start refer to the bottom left-hand corner of the box that one may imagine enclosing the text, rather than the top left-hand corner as is the case when printing text following a VDU 5.

Plot Type Options

The plot type parameter contains four flags whose meanings are as follows:

Bit 0: Set – Justify Text
When this bit is set, the Font Painter will attempt to fully justify the text string. To do this, you must have supplied the co-ordinates of the right-hand limit to which justification should extend. Only the X-axis part of this co-ordinate is really relevant since the Y-axis value must be the same as that of the text starting position if you want horizontal text! The co-ordinate is supplied by issuing a graphics MOVE command before issuing the SWI, so for example:

```
MOVE 1280,732
SYS "Font_Paint",,"This will be justified",%10001,0,732
```

will produce the text string "This will be justified" with the lower left-hand corner of the opening 'T' at (0,732) and the lower right-hand corner of the closing 'd' at (1280,732).

Bit 1: Set – Pre-draw Rubout Box
This option tells the Font Painter to 'rub out' a box surrounding the text by filling it with the background colour before the text is painted onto the screen. To do this, two pairs of co-ordinates must have been supplied by issuing MOVE commands: the first to specify the bottom left-hand corner of the box and the second the top right-hand corner. For example:

```
MOVE 0,716:MOVE 1280,772
SYS "Font_Paint",,"A rub out box",Trinity%,%10010,0,732
```

If this operation is to be combined with justification the box co-ordinates must be issued first, followed by the justification limit co-ordinate. For example:

```
MOVE 0,716:MOVE 1280,772 :REM rub out box
MOVE 1280,732
SYS "Font_Paint",,"This will be justified",%10011,0,732
```

Bit 2: Set – Absolute Co-ordinates Supplied
This bit *should* allow co-ordinates to be specified relative to the cursor position or absolutely (when set). OS v1.20 ignores the state of this bit and always treats the co-ordinates as absolute.

Bit 3: Not used

Bit 4: OS Co-ordinates/Font Painter Co-ordinates
This bit allows the co-ordinates of the text display position to be set either in OS units (ie, within the theoretical display limits of 1280,1024) or in Font Painter co-ordinates with units of $^1/_{72000}$ths of an inch. The application of the latter form is limited, so it is suggested you always set this bit and use OS co-ordinates (as we have).

Conversions Between the Co-ordinate Systems

Two SWIs are provided to convert between Font Painter co-ordinates in $^1/_{72000}$ths of an inch and OS VDU driver co-ordinates. These SWIs simply take an X,Y co-ordinate pair in R1,R2 and return its converted form in R1,R2.

SWI Font_ConverttoOS (SWI &40088)
Convert Font Painter to OS

This SWI converts a pair of Font Painter co-ordinates in units of $^1/_{72000}$th inch to OS co-ordinates. On entry, the X,Y pair is supplied in R1,R2 with the converted result being returned in the same registers.

SWI Font_Converttopoints (SWI &40089) Convert OS to Font Painter

This SWI converts a pair of OS co-ordinates in screen units to Font Painter co-ordinates in $^{1}/_{72000}$th inch units. On entry, the X,Y pair is supplied in R1,R2 with the converted result being returned in the same registers.

Size Calculations for Characters and Strings

The Font Manager provides several SWIs to allow detailed information about the currently selected font to be examined. These are documented below:

SWI Font_ReadDfn(SWI &40083) Read Font Definition

This call returns a set of registers containing information about the font whose handle is supplied. On entry R0 should contain a font handle and R1 should point to a buffer large enough to hold the font name (<= 12 bytes). The results are returned in registers as follows:

Register	Information
R1	Still points to buffer, which now contains font name
R2	Width of font *16
R3	Height of font *16
R4	Width resolution in pixels per inch
R5	Height resolution in pixels per inch
R6	Number of other users of font
R7	Number of other accesses since this font was last used

SWI Font_ReadInfo (SWI &40084) Read Character Information

This call returns the co-ordinates of a box which is just large enough to accommodate any character in the font, useful for deciding how large a rub out box to define. On entry, R0 must contain the font handle, with the co-ordinates returned as follows:

Register	Information
R1	X_min (OS co-ordinates, inclusive)
R2	Y_min (OS co-ordinates, inclusive)
R3	X_max (OS co-ordinates, exclusive)
R4	Y_max (OS co-ordinates, exclusive)

SWI Font_StringWidth (SWI &40085)
Calculate String Width in Current Font

This call performs a number of calculations on a text string to determine how much space it will occupy.

On entry R1 must point to the string, R2 and R3 specify the maximum offsets in Font Manager co-ordinates which the string is allowed to occupy (see below), R4 contains the character at which the string may be split if it will not fit entirely and R5 contains the positional index of the last character allowed in the calculation.

The results returned indicate where the cursor would be after the string was printed as (X,Y) in R2 and R3, the number of occurrences of the specified split character in R4 and the positional index of the point at which the calculation terminated (according to the character criteria supplied).

By judicious use of these parameters it is possible to determine where to split text lines for wordprocessor formatting (by setting R4 to ASCII 32), find the 'length' of a string or calculate proportional spacing values.

To show this facility in action, the example program right-justifies a piece of text by:

1. Defining a rub out box.
2. Converting its size into Font Manager co-ordinates.
3. Calculating the 'length' of the string in Font Manager co-ordinates.
4. Converting these co-ordinates back into OS co-ordinates.
5. Printing the string at an X offset of 1280 minus the previous result.

The following program fragment reproduces this in much the same way as the example listing 14.1:

```
MOVE 0,460:MOVE 1280,600:REM set up rub out box
SYS "Font_Converttopoints",,1280,140 TO ,BoxXinPts%,BoxYinPts%
SYS "Font_StringWidth",,"Try Corpus",BoxXinPts%,BoxYinPts%,32,13
TO,,XoffPts%,YoffPts%  SYS "Font_ConverttoOS" ,,XoffPts%,YoffPts%
TO ,Xoff%,Yoff%  SYS "Font_Paint",,"Try Corpus",%10010,1280-
Xoff%,500
```

Conclusion

The Font Manager and Font Painter provide the basis for complex text display facilities based on high-resolution, anti-aliased fonts. Unfortunately, the complexity of the font management software as a whole makes the process of executing software written for this module rather painful to watch. Furthermore, anti-aliasing only works at its best for relatively large characters displayed in black on white, which is not really surprising given the heritage of the technique in the world of typesetting.

```
 10  REM >List14/1
 20  REM by Nicholas van Someren
 30  REM Archimedes OS: A Dabhand Guide
 40  REM (c) Copyright AvS and NvS 1988
 50  :
 60  MODE 12
 70  :
 80  REM The variable Font$Prefix should be set to
 90  REM the directory containing the fonts.
100  :
110  *SET Font$Prefix $.Welcome.Fonts
120  :
130  REM Read Font Manager version and load fonts.
140  :
150  SYS "Font_CacheAddr" TO version%,cacheused%,cachesize%
160  vers$="Font Manager vers. "+STR$(version%/100)
170  SYS "Font_FindFont",,"Trinity.Medium",48*16,48*16 TO
Trinity%
180  SYS "Font_FindFont",,"Corpus.Medium",32*16,32*16 TO Corpus%
190  SYS "Font_FindFont",,"Trinity.Medium",24*16,18*16 TO
SmallTrinity%
200  :
210  REM Set up the palette and show them as stripes.
220  :
230  SYS "Font_SetPalette",,0,1,6,&00000000,&F0F0F000
240  SYS "Font_SetPalette",,7,8,7,&F0F0F000,&00600000
250  FOR s%=0 TO 1279 STEP 80
260  GCOL s%/80
270  RECTANGLE FILL s%,0,79,1023-32
280  NEXT
290  :
300  REM Choose the colours and the Trinity font.
310  :
320  SYS "Font_SetFontColours",Trinity%,7,8,7
330  SYS "Font_SetFont",Trinity%
340  :
350  REM Define a rub out box and print a message.
360  :
370  MOVE 50,800
380  MOVE 1230,940
390  SYS "Font_Paint",,vers$,%10010,50,840
```

```
  400  :
  410  REM Set up a rub out box, and the cursor
  420  REM position after justification.
  430  :
  440  MOVE 100,516:MOVE 1180,572
  450  MOVE 1180,532
  460  :
  470  REM Select the font and display a message.
  480  :
  490  SYS "Font_SetFont",SmallTrinity%
  500  SYS "Font_Paint",,"This is 24pt justified Trinity",
%10011,100,532
  510  :
  520  REM Select some new colours and the Corpus font.
  530  :
  540  SYS "Font_SetFontColours",Corpus%,0,1,6
  550  SYS "Font_SetFont",Corpus%
  560  :
  570  REM Define a rub out box
  580  :
  590  MOVE 0,160
  600  MOVE 1280,300
  610  :
  620  REM Find the point size of the rub out box and
  630  REM the string size (not exceeding the box).
  640  :
  650  text$="This is on the right"
  660  SYS "Font_Converttopoints",,1280,140 TO ,BoxXinPts%,
BoxYinPts%
  670  SYS "Font_StringWidth",,text$,BoxXinPts%,BoxYinPts%,32,
LEN(text$) TO ,,XoffPts%,YoffPts%
  680  :
  690  REM Convert the size back to OS units and print
  700  REM some right-justified text.
  710  :
  720  SYS "Font_ConverttoOS",,XoffPts%,YoffPts% TO ,Xoff%,Yoff%
  730  SYS "Font_Paint",,text$,%10010,1280-Xoff%,208
  740  END
```

Listing 14.1. Font demonstration.

15 : Sound Introduction

The Archimedes A-series machines support an eight channel, stereo digital sound system. This is provided by VIDC using a Direct Memory Access (DMA) output channel and analogue output circuitry (DAC). There is no analogue sound system on the Archimedes. All sounds, or waveforms, are created either mathematically or by use of data tables. The sound data is processed entirely by software which must also perform any filtering or modulation required.

The DMA generates audible sounds by having the digital data held in its buffer converted into analogue signals. The buffer itself is filled by the Voice Generators which are normally implemented as Relocatable Modules and have direct access to the sound channels.

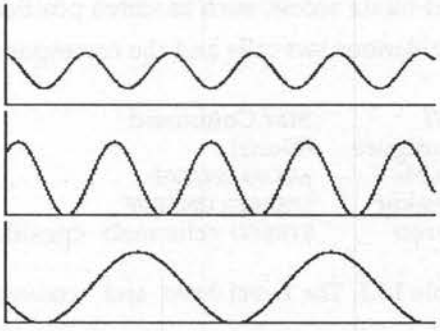


Figure 15.1. Graphs

The data itself is first logarithmically scaled to provide a greater dynamic range. This scaling is undertaken automatically by the firmware. The value of each piece of data can be thought of as an indication of the position of a speaker cone. For example, a constant series of values will not produce any sound at all as the speaker cone will not move. On the other hand a rapidly fluctuating series of values will produce a high frequency tone. The greater

the fluctuation of the series the louder the tone. Likewise a series which fluctuates only a little would produce a quiet tone.

The Three Levels of Sound

The Archimedes sound system is split up into three levels which are hierarchically structured. The lowest level is the DMA system, this in turn activates higher levels which then provide a block of sound data for output to the DAC.

Level 0 – SoundDMA

The DMA Buffer Handler is activated whenever a new block of sound data is required, and updates the necessary pointers. Level 0 is essential to the sound system and so it is provided in firmware as a Relocatable Module called SoundDMA. If a sound overload occurs, ie, if there is simply to much sound data to be processed, Level 0 marks the offending channel as over-run and the real-time buffering is aborted, before being restarted.

Level 0 also caters for any hardware-dependent programming which requires privileged-mode access, such as stereo positioning.

Table 15.2 lists the various SWI calls and the corresponding * commands, if one exists.

SWI	Star Command
Configure	(None)
Enable	AUDIO ON/OFF
Speaker	SPEAKER ON/OFF
Stereo	STEREO <channel> <position>

Table 15.1. The Level 0 SWI and * commands.

Level 1 – SoundChannels

The DAC recognises eight logical sound channels and these will produce sound according to the configuration (ie, stereo-position) of each logical channel.

There can be one, two, four or eight physical channels. The number in use will affect the way that the sound buffers are filled, which is accomplished by means of interleaving the data when filling the DMA buffer.

Level 1 controls the allocation and de-allocation of channels and will automatically flush de-allocated channels. Level 1 also provides the means by which pitch and volume are altered, will attach channels to voice generators, install and detach voice generators and allows direct real-time control of musical parameters.

Table 15.2 lists the various SWI calls and corresponding * commands associated with Level 1 sound.

SWI	Star Command
AttachNamedVoice	*CHANNELVOICE <channel> <voice name>
AttachVoice	*CHANNELVOICE <channel> <voice index>
Control	*SOUND<channel><amp><pitch><duration>
ControlPacked	
InstallVoice	*VOICES
LogScale	
Pitch	
ReadControlBlock	
RemoveVoice	
SoundLog	
Tuning	*TUNING <n>
Volume	*VOLUME <n>
WriteControlBlock	

Table 15.2. SWI calls associated with Level 1 sound

Level 2 – SoundScheduler

Level 2 facilitates the queuing of sounds, ie, their playing in a predetermined order. In addition it also provides the data structures which allow multiple channel music or sound to be synchronised under simple program control – the playing of chords.

Notes, timbral changes and most importantly user-supplied code routines may be scheduled in arbitrary time order. Queued sounds are activated as events at the appropriate tempo-dependent time in the future. Unfortunately it is not possible to queue sounds to play in the 'past', as some sound systems allow. Level 2 also allows changes in tempo and beats per bar to change dynamically whilst maintaining note synchronisation.

Table 15.3 (overleaf) lists the various SWI calls and corresponding * commands associated with Level 2 sound.

SWI	Star Command
QBeat	
QDispatch (reserved)	
QFree	
QInit	
QRemove (Reserved)	
QSchedule	*QSOUND<chan><amp><pitch><dur><nBeats>
QTempo	*TEMPO <n>

Table 15.3. Level 2 SWI calls.

16 : Sound Star Commands

This chapter details the various * commands associated with sound and their use. They are classified in alphabetical order under their sound level num-ber as defined in the previous chapter.

Level 0 Commands

*AUDIO

Syntax:

```
*AUDIO ON/OFF
```

All sound interrupt and DMA activity is stopped when *AUDIO OFF is issued. The DMA buffer is no longer filled so no sound is produced by the DAC. Once Audio is turned back on with *AUDIO ON, the DMA and interrupt system are returned to the status they held prior to being turned off. During the time when audio is off, Level 1 and Level 2 activities are also suspended, although software interrupts to all levels are still permitted, even if no sound results. This can be advantageous as it allows sounds to be 'remembered' while AUDIO is off, and to be hastily played as soon as AUDIO is turned back on.

*SPEAKER ON/OFF

Syntax:

```
*SPEAKER ON/OFF
```

This command effectively turns the internal speaker on or off – it has no effect on the external stereo output. It achieves this by muting the monophonic mixed signal to the internal loudspeaker amplifier. All DMA activity continues so sounds will be processed as usual. If you do not envisage using the external stereo output, then this command can be used instead of the AUDIO command.

*STEREO

Syntax:

```
*STEREO <channel> <position>
```

where:

<channel> is in the range 1 to 8
<position> is in the range –127 to 127.

The DAC supports stereo sound output via the audio socket at the rear of the machine. Each logical channel, therefore, has its own unique stereo image. This command usually sets the stereo position of each physical channel, but can be used to reposition the logical channels if used correctly.

The channels affected are n, n+N, n+2N up to channel 8, where N is the number of active voices. The default stereo settings for all channels is zero, ie, centre. There are seven discrete stereo positions:

Range	Position
–127 to –80	Full Left
–79 to –48	2/3 left
–47 to –16	1/3 left
–15 to +15	Central
+16 to +47	1/3 right
+48 to +79	2/3 right
+80 to +127	Full right

Level 1 Commands

*CHANNELVOICE

Syntax:

```
*CHANNELVOICE <channel><voice index>/<voice name>
```

where:

<channel> is in the range 1 to 8
<voice index> is in the range 1 to 32.

This command allows different physical channels to be attached to different VoiceGenerators, in order for them to play different sounds. The voice index is the VoiceGenerator slot number. This can be determined using the *VOICES command. The voice name is the VoiceGenerator's preferred name, which is registered with the Level 0 handler.

Examples of use:

```
*CHANNELVOICE 1 1
*CHANNELVOICE 1 StringLib-Soft
```

*CONFIGURE SoundDefault

Syntax:

```
*CONFIGURE SoundDefault<speaker><coarsevol><voice>
```

where:

<speaker> is 0 or 1
<coarsevol> (coarse volume) is in the range 0 to 7
<voice> is in the range 1 to 16.

This command sets the default sound configuration. As these settings are held in non-volatile CMOS RAM, they remain in use until the machine is completely reset or they are changed again. Upon power-up, CTRL-RESET, or BREAK, these values are used to initially configure the speaker, master volume control and the default bells' voice.

The speaker parameter controls whether the internal loudspeaker should be on (1) or off (0). The coarse volume sets the default amplitude of the system sound. This value is equivalent to the *VOLUME setting divided by 16, ie, a coarse volume of 2 is equivalent to a *VOLUME setting of 32. The Voice parameter can be used to set which voice generator channel one, the default system bell channel, will be attached to.

Example of use:

```
*CONFIGURE SoundDefault 1 6 7
```

*SOUND

Syntax:

```
*SOUND <channel> <amplitude> <pitch> <duration>
```

This is the direct equivalent of BASIC's SOUND command. When used, and providing the selected channel is active and attached, the sound will be reproduced immediately.

The parameters passed to *SOUND, are all unsigned integers. It should be noted that when passing negative numbers all 32 bits should be passed, for example, to pass the value –1 the equivalent hexadecimal number, &FFFFFFFF, must be passed.

Amplitude
There are two forms of amplitude – linear and logarithmic. The linear form is expressed as a simple number in the range 0 (silence) to –15 (loud). The logarithmic scale runs from &100 (silence) to &17F (loud), and a change of &10 represents a doubling or halving of the volume. Bit seven can be used as a toggle to facilitate a 'smooth' change of the sound.

For example:

```
*SOUND 1 &17F &4200 &FE
*SOUND 1 &1EF &4000 &20
```

will cause the first sound to change pitch and amplitude as soon as the second command is issued.

Pitch
There are two ways in which pitch can be specified. The first method allows specification in steps of a quarter of a semitone. The range of values for this method is 0 to 255. The lowest note (0) is the B, one octave and a semitone below middle C. Middle C has a value of 53. Table 16.1 can be used to look-up the pitch value associated with which note.

Note	1	2	3	4	5	6
A		41	89	137	185	223
A#	1	45	93	137	189	237
B	5	49	97	141	193	241
C	9	53	101	145	197	245
C#	13	57	105	149	201	249
D	17	61	109	153	205	253
D#	19	65	113	161	209	
E	21	69	117	165	213	
F	25	73	121	169	217	
F#	29	77	125	173	221	
G	33	81	129	177	225	
G#	37	85	133	181	229	

Table 16.1. Note associated Pitch values.

A finer degree of control can be obtained using a second method in which values are specified in the range &0100 to &7FFF. Each value is constructed from 15-bits. The top three bits (14, 13 and 12) specify the octave number, while the remaining 12-bits define the fractional part of the octave. Each octave is, therefore, split up into 4096 different pitch levels. Middle C in this case has a value of &4000. Table 16.2. defines these values for each note for ease of reference:

Note	1	2	3	4	5	6	7	8	9
A		&0C00	&1C00	&2C00	&3C00	&4C00	&5C00	&6C00	&7C00
A#		&0D55	&1D55	&2D55	&3D55	&4D55	&5D55	&6D55	&7D55
B		&0EAA	&1EAA	&2EAA	&3EAA	&4EAA	&5EAA	&6EAA	&7EAA
C		&1000	&2000	&3000	&4000	&5000	&6000	&7000	
C#	&0155	&1155	&2155	&3155	&4155	&5155	&6155	&7155	
D	&02AA	&12AA	&22AA	&32AA	&42AA	&52AA	&62AA	&72AA	
D#	&0400	&1400	&2400	&3400	&4400	&5400	&6400	&7400	
E	&0555	&1555	&2555	&3555	&4555	&5555	&6555	&7555	
F	&06AA	&16AA	&26AA	&36AA	&46AA	&56AA	&66AA	&76AA	
F#	&0800	&1800	&2800	&3800	&4800	&5800	&6800	&7800	
G	&0955	&1955	&2955	&3955	&4955	&5955	&6955	&7955	
G#	&0AAA	&1AAA	&2AAA	&3AAA	&4AAA	&5AAA	&6AAA	&7AAA	

Table 16.2. Table of Pitch values.

*TUNING

Syntax:

```
*TUNING n
```

where:

n is a value in the range –&OFFF to &OFFF

This command overrides the system pitch base. It can be used to raise or lower the pitch of all sounds. The value is a 16-bit number, which is split up into two parts.

The top four-bits represent the octave number (O), the remaining bits are the fractional part of the octave (FFF). A value of zero resets the tuning to the default value. The value is relative, so to raise the pitch of all notes by one octave, you would use:

```
*TUNING &1000
```

*VOICES

This command will list all the currently installed VoiceGenerator's names which have been registered with the Level 0 handler. Channels that have been attached to these voices are also indicated. The voice number is the voice index which is used in the *CHANNELVOICE command. The numbering of the voices depends on the order in which they were installed, so it may change. Typically the command might return the following:

```
          Voice    Name
1         1        WaveSynth-Beep
 2        2        StringLib-Soft
  3       3        StringLib-Pluck
    4     4        StringLib-Steel
          5        StringLib-Hard
          6        Percussion-Soft
          7        Percussion-Medium
          8        Percussion-Snare
          9        Percussion-Noise

^^^^^^^^Channel Allocation Map
```

Figure 16.1. Default *VOICES output.

*VOLUME

Syntax:

```
*VOLUME n
```

where:

n is a value in the range 1 to 127.

This command sets the master volume of the sound system. The range is 1 (quietest) to 127 (loudest). This value is used by all voice generators to scale the amplitudes of their sounds. The amplitude of any sound command will be scaled by this value. The default value is 127 and this may be changed by using the *CONFIGURE VOLUME command. For example:

```
*VOLUME 63
```

or alternatively:

```
*CONFIGURE VOLUME 63
```

Level 2 Commands

*QSOUND

Syntax:

```
*QSOUND <chan> <amp> <pitch> <duration> <nBeats>
```

This command is similar to the *SOUND command, but differs in that it is possible to specify on which beat this note should be played. For full information on the first four parameters see the description for *SOUND above.

The beat is an internal counter which is set to zero at the start of each bar. The beat increment is set by the *TEMPO command. If a value of &FFFFFFFF (–1) is used for nBeats then, instead of being scheduled for a given number of beats, the sound is synchronised with the last scheduled sound. For example:

```
10 BEATS 200
20 REPEAT
30 UNTIL BEAT =0 :REM Wait for start of next bar
40 *QSOUND 1 &17F &4000 &10 50
50 *QSOUND 2 &17F &5000 &10 &FFFFFFFF
```

This program will produce two tones, one octave apart, to be made 50 beats after the start of the next bar.

*TEMPO

Syntax:

```
*TEMPO n
```

where n is a value in the range 0 to &FFFF

This command affects the rate at which scheduled events are played back. The value n, in fact, sets the rate at which the beat counter is incremented, and is used to queue the scheduled sounds, for instance:

Value	Beat increment	Scheduled sounds
2048(&0800)	Each beat lasts twice as long	Slowed down
4096(&1000)	Default.	
8192(&2000)	Each beat is half as long	Speeded up

The default value is &1000. This corresponds to one microbeat per centi-second.

Each bar is split up into a number of beats. The duration of each beat is affected by the tempo. The beat counter is set to zero at the start of each bar. If sounds or music are scheduled using *QSOUND or the five parameter BASIC SOUND statement, then the execution of notes can be speeded up or slowed down by changing the TEMPO. The durations of the notes are not changed.

17 : Sound SWI Calls

The SWI calls associated with the Archimedes sound system are detailed below. They are arranged according to their level classifications.

Level 0 SWI Commands

Sound_Configure

Pass:

R0 Number of channels (n) [rounded up to 1,2,4,8 (N)]
R1 Samples per buffer
R2 Period in microseconds (μS)
R3 Level1 handler code. [normally 0 to preserve system level1]
R4 Level2 handler code. [normally 0 to preserve system level2]

Zero for any parameters will not change that setting.

Constraints:

R0 1 .. 8 (rounded to 1,2,4,8)
R1 16/N .. Sound DMA Buffer limit/N
R2 3/N .. 255/N

Returned:

Previous settings.

Default:

R0 1
R1 &D0
R2 48

The Number of Sound Channels (R0)

The value passed in R0 will be rounded to 1,2,4,8. Channels are multiplexed into the eight logical channels available. If only one physical channel is available then all eight logical channels will be used for sound. Similarly if only two physical channels are available, then every other logical channel will be used for each physical channel, ie, channel 1 will use logical channels 1,3,5 and 7, and channel 2 will use logical channels 2,4,6 and 8.

Because of this interleaving the channels are multiplexed into one half, quarter or eighth of the sample period. This results in the overall signal level per channel being scaled down by the same amount, so the overall signal peak level for all multi-channel modes remains constant. The more physical channels in operation, the quieter each one becomes.

The DMA Buffer Size and Sample Rate (R1, R2)

As a rule these settings should not generally be altered. The sample rate is probably of greatest use as its parameter defines how long each piece of sample data should last. 10000 bytes of sample data played at the default sample rate of 48 microseconds would result in a sound lasting 0.48 seconds. If the sample rate was then changed to 50 microseconds, the sound would last 0.50 seconds. Therefore, if data is sampled at different sample rates, but the same relative pitch needs to be maintained, then this can be achieved by altering the sample rate value. The value of 48 microseconds is derived in the following way:

8 channel multiplex rate	=	166.666 kHz
Overall audio sample rate	=	20.833 kHz
period per byte	=	1/sample rate = 1/(20833)
	=	4.8e-5
	=	48 µsec

The DMA buffer size should not be changed. Every time the buffer becomes empty a buffer fill request interrupt occurs, and the voice generators are requested to fill the buffer. It is important that this interrupt rate is kept at a sensible level so as not to interrupt the processor to often. The default rate is around one centisecond – more precisely 0.99841 centiseconds. This value of 0.99841 centiseconds is derived in the following way:

Sample rate per channel	20833 Hz	
Bytes per channel (DMA size)	208	
Interrupt rate	20833/208	= 100.1587 Hz
Buffer period	1/100.1587	= 0.99841 centiseconds

The following program (Listing 17.1) demonstrates the use of the Sound_Configure SWI; the effect of more voices on the amplitude and the effect of changing the microseconds per sample has on pitch.

```
10  REM >List17/1
20  REM (C) Felix Andrew 1988
30  REM Archimedes OS: A Dabhand Guide
40  REM
50  REM Example sound programs.
60  :
70  MODE 8
80  PROCrestore
```

```
  90  PRINT"Sound_Configure"'
 100  :
 110  SYS "Sound_Configure",1
 120  PROCsound_info
 130  PRINT'"Configured for one voice"
 140  SOUND 1,&17F,200,20
 150  PRINT"SOUND 1,&17F,200,20 generates a loud hi pitched sound"
 160  PROCpause
 170  :
 180  SYS "Sound_Configure",8
 190  PROCsound_info
 200  PRINT'"Configured for EIGHT voices"
 210  SOUND 1,&17F,200,20
 220  PRINT"The same sound statement produces a much quieter
sound"
 230  PROCpause
 240  :
 250  SYS "Sound_Configure",1,0,50
 260  PROCsound_info
 270  SOUND 1,&17F,200,20
 280  PRINT'"The microsecond per sample has now been increased."
 290  PRINT"This results in a tone of lower pitch."
 300  PRINT"This is because every piece of data is assumed to take
more time"
 310  PRINT"to process, ie, the PLAY back rate has been lowered."'
"or the sample rate increased."
 320  A=INKEY(20*5)
 330  END
 340  :
 350  DEF FNnum(L%,V%)
 360  =RIGHT$(STRING$(L%," ")+STR$V%,L%)
 370  :
 380  DEF PROCsound_info
 390  SYS "Sound_Configure",0,0,0,0 TO N,SPB%,US%,L1%,L2%
 400  PRINT"Channels       ";FNnum(2,N)
 410  PRINT"uS per sample ";FNnum(2,US%)
 420  ENDPROC
 430  :
 440  DEF PROCpause
 450  A=INKEY(20*5)
 460  PRINT"Press any key":A=GET
 470  PRINT
 480  ENDPROC
 490  :
 500  DEF PROCrestore
 510  *TUNING 0
 530  ENDPROC
```

Listing 17.1. Demonstrating the Sound_Configure SWI.

Sound_Enable

Passed:

R0 New State
2 = ON
1 = OFF
0 = No change (just read previous state)

Returned:

R0 Previous state
2 = ON
1 = OFF

This SWI is used by the *AUDIO command and suspends any DMA interrupts. It is important to remember that any subsequent sound commands may be 'stored', so that when the sound is enabled the 'stored' sounds will be played immediately.

There is no simple way to flush the sound buffers. The only way to clear the sounds is to detach and then re-attach the voices, or to set the SCCB channel flags for each to two, which will make the Level 0 handler flush the channels, however, doing this is prone to problems, as the Voice Generator's will not be informed of the change.

Sound_Speaker

Passed:

R0 New State
2 = ON
1 = OFF
0 = No change (just read previous state)

Returned:

R0 Previous state
2 = ON
1 = OFF

This SWI is used by the *SPEAKER command and switches off the mono-mix of the left and right audio channels to the internal loudspeaker amplifier. Sounds are still processed and DMA interrupts continue. The stereo audio output remains active. The following program (Listing 17.2) illustrates the 'storing' of sounds while Audio is off and the effect of Speaker off.

```
  10  REM >List17/2
  20  REM (c) Felix Andrew 1988
  30  REM Archimedes OS: A Dabhand Guide
  40  REM
  50  REM Example sound programs.
  60  :
  70  VOICES 1
  80  *CHANNEL. 1 1
  90  :
 100  MODE 8
 110  PRINT"SWI Sound_Enable"'
 120  PROCspeaker(%10):PROCstate(%10):SOUND 1,-15,100,10
 150  PRINT"SOUND 1,-15,100,10"'"Press a key":A=GET:PRINT
 160  PROCstate(%01):SOUND 1,-15,100,10
 161  PRINT"SOUND 1,-15,100,10":PRINT"Press a key":A=GET:PRINT
 200  PROCstate(%10)
 210  PRINT"Sound statements issued when the sound is disabled are
produced"
 220  PRINT"when the sound is then turned on."
 230  PRINT"Buffers unfortunately cannot easily be flushed."
 240  PRINT"To flush the channels, detach all voices and the
reattach them."
 250  A=GET:PROCstate(%01)
 260  :
 270  CLS
 280  PRINT"SWI Sound_Speaker"'
 290  PROCstate(%10):PROCspeaker(%10):SOUND 1,-15,100,10
 320  PRINT"SOUND 1,-15,100,10"'"Press a key":A=GET:PRINT
 321
 330  PROCspeaker(%01):PRINT"SOUND 1,-15,100,10"
 350  SOUND 1,-15,100,10
 360  PRINT"Sounds played while the speaker is off are still
played"
 370  PRINT"Press a key":A=GET:PRINT
 380  PROCspeaker(%10)
 390  END
 400  :
 410  DEF PROCstate(V%)
 420  SYS "Sound_Enable",V%
 430  SYS "Sound_Enable",0 TO R%
 440  IF R%=1 THEN PRINT"Sound   Off" ELSE PRINT"Sound   On"
 450  ENDPROC
 460  :
 470  DEF PROCspeaker(V%)
 480  SYS "Sound_Speaker",V%
 490  SYS "Sound_Speaker",0 TO R%
 500  IF R%=1 THEN PRINT"Speaker Off" ELSE PRINT"Speaker On"
 510  ENDPROC
```

Listing 17.2. Storing sounds.

Sound_Stereo

Passed:

R0 Logical / Physical Channel
R1 Image position

-128	no change
-127	Full left
-79	2/3 left
-47	1/3 left
0	Central
47	1/3 right
79	2/3 right
127	Full right

Returned:

R0 Preserved
R1 Previous image position

This SWI facilitates the re-positioning of the stereo image of the specified channel. There are seven stereo image positions. Depending on the number of physical channels, the call can be used to either position the physical OR logical channels. Repositioning of the logical channels is not advised.

For N physical channels enabled this call will move channels C, C+N, C+2N up to logical channel 8. A table of the logical channels that will be moved according to the physical channel specified and the number of physical channels is given below:

Physical	Channel	Logical channels moved
1	1	1,2,3,4,5,6,7,8
	2	2,3,4,5,6,7,8
	3	3,4,5,6,7,8
	4	4,5,6,7,8
	5	5,6,7,8
	6	6,7,8
	7	7,8
	8	8
2	1	1,3,5,7
	3	3,5,7
	5	5,7
	7	7

Physical	Channel	Logical channels moved
	2	2,4,6,8
	4	4,6,8
	6	6,8
	8	8
4	1	1,5
	5	5
	2	2,6
	6	6
	3	3,7
	7	7
	4	4,8
	8	8
8	1	1
	2	2
	3	3
	4	4
	5	5
	6	6
	7	7

This method of moving the logical channels can be used to produce different stereo positions.

The following program (Listing 17.3) indicates the relative merits of logical channel stereo positioning, as opposed to physical channel positioning and positioning of the channels while a sound is being played. For sensible results this program is best run using speakers or headphones attached to the audio jack at the rear of the machine.

```
  10  REM >List17/3
  20  REM (c) Felix Andrew 1988
  30  REM Archimedes OS: A Dabhand Guide
  40  REM
  50  REM Example sound programs.
  60  :
  70  MODE 8
  80  PRINT"SWI Sound_Stereo"'
  81  PRINT"This effect can not be experienced by using the
internal speaker"'
  90  DIM posn(7)
 100  FOR N%=1 TO 7
 110  READ posn(N%)
 120  NEXT
 130  :
 140  VOICES1
```

```
150 PRINT"Moving stereo posn THEN playing a sound"
160 SYS "Sound_Stereo",1,0:SOUND 1,-15,1,10
170 FOR P%=1 TO 7
180 PRINT"Stereo ";posn(P%)
190 SYS "Sound_Stereo",1,posn(P%)
200 SOUND 1,-15,100,10
210 A=INKEY(40)
220 NEXT
230 :
240 PROCpak
250 PRINT"Moving stereo posn WHILE playing a sound"
260 SOUND 1,-15,100,10*6
270 FOR P%=7 TO 1 STEP-1
280 PRINT"Stereo ";posn(P%)
290 SYS "Sound_Stereo",1,posn(P%)
300 A=INKEY(40)
310 NEXT
320 PRINT"The click occurs when the stereo posn is moved"
330 PROCpak
340 CLS
350 :
360 PRINT"Moving the logical channels"'
370 VOICES2
380 PRINT"Logical channels 1,3,5 and 7 on the left"
390 SYS "Sound_Stereo",1,-127:SOUND 1,-15,180,10:PROCpak
400 :
410 PRINT"Logical channel 7 on the right"
420 SYS "Sound_Stereo",7,127:SOUND 1,-15,180,10:PROCpak
430 :
440 PRINT"Logical channels 5 and 7 on the right"
450 SYS "Sound_Stereo",5,127:SOUND 1,-15,180,10:PROCpak
460 :
470 PRINT"Logical channels 3,5 and 7 on the right"
480 SYS "Sound_Stereo",3,127:SOUND 1,-15,180,10:PROCpak
490 :
500 PRINT"Logical channels 1,3,5 and 7 on the right"
510 SYS "Sound_Stereo",1,127:SOUND 1,-15,180,10:PROCpak
530 END
540 :
550 DATA -127,-79,-47,0,47,79,127
560 :
570 DEF PROCpak
580 PRINT"Press a key":A=GET:PRINT
590 ENDPROC
600 :
610 DEF FNnum(L%,V%)
620 =RIGHT$(STRING$(L%," ")+STR$V%,L%)
630 :
640 DEF FNhnum(L%,V%)
650 =RIGHT$(STRING$(L%," ")+STR$~V%,L%)
```

Listing 17.3. Stereo re-positioning.

Level 1 SWI Commands

Sound_InstallVoice

Passed:

R0 Voice Generator Header Code
0 for don't change

R1 Voice slot specified
0 for next free slot

Returned:

R0 String pointer – name of previous voice (or error message)

R1 voice number allocated
0 indicates a failure to install

This SWI is used to install a voice generator at which point any number of physical channels can be attached to it. The code necessary to produce voice generators is given below. The installed voice list can be examined by passing zero in R0, on exit the voice name will be pointed to by R0. In Arthur 1.2 the voice table is limited to 32 entries.

Sound_RemoveVoice

Passed:

R1 Voice slot to remove

Returned:

R0 string pointer – name of previous voice (or error message)
R1 voice number removed
0 indicates a failure to remove a voice

This SWI is used to remove voice generator's. It may be called if the voice generator is a module and the RMA is tidied, or cleared.

Sound_AttachNamedVoice

Passed:

R0 Physical channel number
R1 Pointer to voice name (zero terminated string)

Returned:

R0 Preserved
R1 Preserved
0 if fail

This call is used by the *CHANNELVOICE command and will attach the physical channel specified in R0 to the named voice. When a channel is attached to a new voice, the previous voice is shut down and the new voice reset. There is no facility to swap voices while a sound is being played. This level will attach the logical channels depending on the number of physical channels.

Sound_AttachVoice

Passed:

R0 Physical channel number
R1 Voice slot to attach
0 to detach voice and mute

Returned:

R0 Preserved
0 if illegal channel number
R1 Previous voice number
0 if not previously attached

This SWI allows a particular physical channel to be attached to a voice slot, the number of which can be ascertained from the output information displayed by *VOICES. Voice slot numbers may change so *VOICES should always be used first to obtain the correct number.

When a new voice is attached, the old voice is first shut down and there is no facility to swap voices mid-sound. This call is used by the *CHANNELVOICE command and this level caters for the logical channels.

Sound_Volume

Passed:

R0 Sound volume
1 (quietest) to 127 (loudest)
0 will inspect last setting

Returned:

R0 Previous volume

This SWI sets the overall volume and is used by the *VOLUME command. Internally the volume is represented as a 7-bit logarithmic value, so a change of &10 represents a doubling or halving of the volume. Voice generators should take note of this value, and scale their waveforms accordingly. The default value is taken from the CMOS RAM setting.

Sound_SoundLog

Passed:
R0 Signed 32-bit number

Returned:
R0 8-bit signed scaled logarithm

This SWI converts 32-bit signed integers to an 8-bit signed logarithmic value using an internal lookup table. It is used to convert sampled data to a logarithmic scaled value for use in Voice Generators.

Sound_LogScale

Passed:
R0 8-bit signed audio logarithm

Returned:
R0 8-bit scaled logarithm

This SWI maps an internal 8-bit signed logarithm to one scaled to the current volume.

Sound_Pitch

Passed:
R0 15-bit pitch value
bits 14-12 represent the octave number
bits 11-0 represent the octave division

Returned:
R0 32-bit phase accumulator value

This SWI maps a 15-bit pitch to an internal format pitch value.

Sound_Tuning

Passed:
R0 New tuning value
0 = don't change

Returned:
R0 Previous setting

This SWI sets the tuning parameter and is used to offset the pitch values used throughout the system and is used by the *TUNING command. See the

description of the *TUNING command in the previous chapter for an explanation of the range of R0.

Sound_Control

Passed:

R0 Physical Channel
R1 Amplitude
Either 0-15
or &100 to &1FF
of which bit 7 is: 0 for GATE ON/OFF
1 for smooth update
R2 Pitch
Either 0-&FF
or &0100 TO &7FFF for enhanced pitch control
bits 14-12 represent octave number
bits 11- 0 represent fractional part of the octave
R3 duration in 5 centisecond periods
except &FF which is infinite

Returned:

R0-R3 Preserved

This SWI command allows immediate execution of the specified sound channel – the parameters take effect on the next buffer fill entry. The GATE on/off causes a new note to be played resulting in a possible 'click', whilst smooth causes the changing of note parameters without restarting the note. This allows the pitch or volume of a note to be altered while it is playing, and can be effectively used for pitch-bend.

For an explanation of amplitude and pitch see the *SOUND command.

Note: GATE is in effect a switch which allows the sound to be turned off (effectively killing the last note) then on (to play the next one). The GATE option should *not* be used if a smooth change from one note to another is required.

Sound_ControlPacked

Passed:

R0 Amplitude and channel (&AAAACCCC)
High word is amplitude, low word is channel
R1 Duration and pitch (&DDDDPPPP)
High word is duration, low word is pitch

Returned:
R0,R1 Preserved

In operation this SWI is similar to Sound_Control, but differs in that its parameters are packed together. For example:

```
SYS "Sound_Control",1,&17F,&4200,16
```

becomes:

```
SYS "Sound_ControlPacked",&017F0001,&00104200
```

Sound_ReadControlBlock

Passed:
R0 Channel
R1 Offset to read from

Returned:
R0 Preserved
0 if fail, invalid read offset
R1 Peserved
R2 32-bit word read if R0 non zero

This SWI will read-32 bit words from the Sound Channel Control Block (SCCB). The values in the SCCB are not standard and they will depend on the particular Voice Generator, and Level 1 handler. The following is a list of the SCCB values which are supposed to be constant under the system Level 1 handler:

Word	Use
0	Amplitude, index to voice table, voice instance, control flags
1	Phase accumulator pitch oscillator
2	
3	Number of buffer fills
4-8	Working registers

Sound_WriteControlBlock

Passed:
R0 Channel
R1 Offset to write
R2 32-bit value to write

Returned:
R0 Preserved
0 if fail, invalid read offset

R1 Preserved
R2 Previous 32-bit word, if R0 non zero

This SWI allows 32-bit words to be written into the SCCB.

Listing 17.4 shows how to attach channels to named and numbered voices. It also contains the rudiments of a BASIC version of the *VOICES command. The use of bit seven in the extended amplitude settings is also demonstrated.

```
 10 REM >List17/4
 20 REM (c) Felix Andrew 1988
 30 REM Archimedes OS: A Dabhand Guide
 40 REM
 50 REM Example sound programs.
 60 :
 70 MODE 8
 80 VOICES1
 90 SYS "Sound_Volume",&7F
100 :
110 PRINT "Sound_AttachNamedVoice"'
120 :
130 PRINT "SYS Sound_AttachNamedVoice,1,StringLib-Hard"
140 SYS "Sound_AttachNamedVoice",1,"StringLib-Hard"
150 SOUND 1,-15,100,10
160 PROChak:CLS
170 :
180 PRINT "SoundInstallVoice"'
190 PRINT "Listing current Installed voices in BASIC"'
200 FOR N%=1 TO 32
210 SYS "Sound_InstallVoice",0,N% TO A$,F%
220 IF F%=0 THEN PRINTA$
230 NEXT
240 PROChak:CLS
250 :
260 PRINT "Sound_AttachVoice"'
270 PRINT "SYS Sound_AttachVoice,1,7"
280 SYS "Sound_AttachVoice",1,7
290 SOUND 1,-15,100,10
300 PROChak:CLS
310 :
320 PRINT"Sound_Control & Sound_Volume"'
330 SYS "Sound_AttachVoice",1,1
340 PRINT"SYS Sound_Control,1,&17F,&4000,&20"
350 SYS "Sound_Control",1,&17F,&4000,&20
360 PROChak
370 :
380 PRINT'"SYS Sound_Volume,&6F":SYS "Sound_Volume",&6F
390 SYS "Sound_Control",1,&17F,&4000,&20
400 PROChak:CLS
410 :
420 SYS "Sound_Volume",&7F
```

```
430  SYS "Sound_AttachVoice",1,7
440  PRINT"Smooth update of sounds"'
450  SYS "Sound_Control",1,&16F,&4000,&20
460  A=INKEY(10)
470  SYS "Sound_Control",1,&1FF,&4200,&20
471  PROChak
480  PRINT"as opposed to"
490  SYS "Sound_Control",1,&16F,&4000,&20
500  A=INKEY(10)
510  SYS "Sound_Control",1,&17F,&4200,&20
520  :
530  END
540  :
550  DEF PROChak
560  PRINT'"Press any key":A=GET
570  ENDPROC
```

Listing 17.4. Attaching channels.

Level 2 SWI Commands

Sound_QInit

Passed:

Nothing

Returned:

R0 0 indicates success

Calling this SWI clears any scheduled sounds queued and resets the tempo and beat variables to their default values.

Sound_QSchedule

Passed:

R0 Schedule period (from start of bar)
&FFFFFFFF (–1) to synchronise with the last schedule event

R1 0 causes a Sound_ControlPacked call
SWI number, of the form &F000000 + SWI number

R2,R3 Are the parameters for R0 and R1 for the SWI

Returned:

R0 0 is successfully queued
<0 for failure, or queue full

This SWI is used by the *QSOUND command. Registers 2 and 3 contain the data which would normally be passed in registers 0 and 1 to the Sound_ControlPacked SWI. Register 0 holds the nBeats parameter. Register 1 will normally be zero, in which case the Sound_ControlPacked SWI will be called, eg,

```
SYS "Sound_QSchedule",10,0,&017F0001,&00204000
```

is equivalent to:

```
SYS "Sound_ControlPacked",&017F0001,&00204000
```

except that it will be played 10 beats after the new bar, when the beat counter is set to zero. This is identical to:

```
*QSOUND 2 &17F &4000 &20 10
```

or alternatively:

```
SOUND 2,&17F,&4000,&20,10
```

which plays a middle C for &20*5 centiseconds on channel 2 when the beat counter reaches 10.

If register 1 is non-zero then other SWIs can be called. This is a very powerful feature. For example, if at the start of the third bar in a piece of music you wanted to attach channel 5 to voice 3, instead of having to continually check to see if the piece of music had reached the third bar, you could simply insert in the sound data the following SWI call:

```
SYS "Sound_QSchedule",0,Sound_AttachVoice,5,3
```

where Sound_AttachVoice is the value of the SWI plus &F000000, ie, Sound_AttachVoice+&F000000.

The program given in listing 17.5 demonstrates the QSchedule command in both forms, by attaching channel one to different voices during the queuing of the sounds.

```
 10  REM >List17/5
 20  REM (c) Felix Andrew 1988
 30  REM Archimedes OS: A Dabhand Guide
 40  REM
 50  REM Example sound programs.
 60  :
 70  MODE 8
 80  PRINT"SYS Sound_QSchedule"
 90  :
100  SYS "Sound_QInit"
110  TEMPO &400:BEATS 100
120  SYS &39,,"Sound_AttachVoice" TO Sound_AttachVoice
```

```
130  Sound_AttachVoice+=&F000000
140  :
150  PRINT"Attach channel one to voice one, ";
160  SYS "Sound_QSchedule",19,Sound_AttachVoice,1,1
170  :
180  PRINT"play a note"'
190  SYS "Sound_QSchedule",20,0,&017F0001,&00104000
200  :
210  PRINT"Attach channel one to voice THREE, ";
220  SYS "Sound_QSchedule",79,Sound_AttachVoice,1,3
230  :
240  PRINT"play a note"'
250  SYS "Sound_QSchedule",80,0,&017F0001,&00104200
260  :
270  END
```

Listing 17.5. Demonstrating the QSchedule command.

Sound_QTempo

Passed:

R0 New tempo 1 to &FFFF (&1000 default)
0 for don't change

Returns:

R0 Previous tempo value

This SWI is used by the *TEMPO command. It is used to set the tempo parameter which is used by the Level 2 scheduler.

Sound_QBeat

Passed:

R0 0 returns beat counter value
–1 returns the current beat COUNT value
< –1 resets the beat counter and COUNT to zero
>0 sets the beat COUNT to N, counts 0 to N–1

Returned:

R0 Current beat counter number is passed
0 otherwise the previous beat COUNT value is used

This call is used by the BASIC commands BEAT and BEATS. The beat counter is an internal counter which starts at zero and counts up to the beat COUNT value.

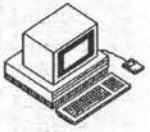
The varied use of QTempo and QBeat are highlighted in the Listing 17.6.

```
  10 REM >List17/6
  20 REM (c) Felix Andrew 1988
  30 REM Archimedes OS: A Dabhand Guide
  40 REM
  50 REM Example sound programs.
  60 :
  70 MODE 8
  80 TEMPO &1000
  90 PRINT"Sound_QBeat"'
 100 :
 110 SYS "Sound_QBeat",&100
 120 PRINT"SWI Sound_QBeat,&100"
 130 PRINT"This has set the beat COUNT to &100, ie beats will
count from"
 140 PRINT"0 to &FF"
 150 PROCinfo
 160 PROChak
 170 :
 180 SYS "Sound_QBeat",-2:PRINT"SYS Sound_QBeat,-2"
 190 PRINT"The beat counter has been turned off"
 200 PROCinfo
 210 PROChak
 220 :
 230 SYS "Sound_QBeat",20:PRINT"SYS Sound_QBeat,20"
 240 PRINT"The beat counter has been turned on, range 0 to 19"
 250 PROCinfo
 260 PROChak
 270 :
 280 CLS
 290 PRINT"Sound_QTempo"'
 300 :
 310 OFF
 320 SYS "Sound_QTempo",&1000
 330 SYS "Sound_QBeat",200
 340 TIME=0:PRINT"At present the counter counts quickly"
 350 REPEAT
 360 SYS "Sound_QBeat",0 TO C%
 370 PRINTTAB(5,5)C%"  ";
 380 UNTIL TIME >500
 390 :
 400 PRINTTAB(0,7)
 410 PRINT"Altering TEMPO to slow it down"
 420 PRINT"SYS Sound_QTempo,&F0"
 430 SYS "Sound_QTempo",&F0
 440 :
 450 TIME=0
 460 REPEAT
 470 SYS "Sound_QBeat",0 TO C%
 480 PRINTTAB(5,11)C%"  ";
 490 UNTIL TIME >500
 500 :
 510 ON:PRINT
 520 END
```

```
530  :
540  DEF PROCinfo
550  SYS "Sound_QBeat",0 TO counter:PRINT"Beat counter ";counter
560  SYS "Sound_QBeat",-1 TO count: PRINT"Beat COUNT   ";count
570  ENDPROC
580
590  DEF PROChak
600  PRINT"Press a key":A=GET:PRINT'
610  ENDPROC
```

Listing 17.6. Using QTempo and QBeat.

18 : Voice Generator



The voice generator is a segment of machine code which is used to fill the DMA buffer on demand. Before it can be used, a generator must first be installed and then channels can be attached to it. The voice generator code contains an eight word entry control block called the Sound Voice Control Block – SVCB.

SVCB entries are used by the Level 1 handler to call the appropriate pieces of code for attaching and detaching the voice under supervisor mode, and real-time buffer filling which is entered in IRQ mode. The title is used by the Level 1 handler to identify the voices and should be concise, but informative.

The speed of the generators is of paramount importance, and so ROM based voice generators are usually copied down into the RMA, as modules, for faster execution of code. Homegrown voice generators should therefore usually be implemented as relocatable modules. This is necessary because if the machine code moves, or is wiped, without the Level 1 handler being informed that the voice is no longer active. Any subsequent use of sound will probably cause an error which, due to the very rapid and repeated calling of the code, will result in a stream of errors which cannot be stopped other than by resetting the machine.

A side effect of this is that modules have to be written in such a way that if the RMA is tidied up the voice will be detached before the tidy, and the reattached afterwards. This is very easily accomplished due to the initialisation and finalisation calls made to a module when such a command is issued.

The SVCB

The SVCB is made up of eight word entries, and their construction is shown overleaf in table 18.1.

Name	Purpose
Sound Voice Fill	Fill DMA buffer
Sound Voice Update	Change SCCB parameters while sound is playing
Sound Voice GateOn	Sets initial parameters for sound
Sound Voice GateOff	Terminates a sound
Sound Voice Instantiate	Channel trying to attach itself to voice
Sound Voice Free	Channel detatched from voice
Sound Voice Install	Channel to be installed
Sound Voice Title	Name given to voice, for level0 handler

Table 18.1. Construction of the SVCB.

The first four entries are used for the real-time DMA buffer filling, and are entered in IRQ mode. Several registers are passed when any of these are called:

R6	Negative if Level 0 configure changed
R7	Channel number
R8	Sample period in microseconds
R9	Pointer to SCCB
R10	DMA buffer limit (+1)
R11	DMA buffer interleave increment
R12	DMA buffer base pointer

The next three entries are used by Level 1 SWI calls, and are therefore entered in Supervisor Mode. The last entry is a relative address offset to the voice name from the start of the SVCB.

Gate On

After a voice has been installed and a channel attached to it, it is ready to generate sound. When a sound command is issued the first piece of code to be called will usually be GateOn. At this point, Register 9 points to the start of the SCCB (Sound Channel Control Block). The SCCB is a 256 byte block of data which contains all the information about a particular channel. The arrangement of this data is listed in table 18.2 overleaf.

Offset	Contents
&00	Amplitude
&01	Index to voice table
&02	Voice instance number
&03	Control / status flags
&04	Phase accumulator pitch oscillator
&08	Timbre, not currently used
&0C	Number of buffer fills
&10	Working Register 4 (Absolute wavetable pointer)
&14	Working Register 5 (Absolute end of wavetable)
&18	Working Register 6 (Absolute pointer to LogAmp table)
&1C	Working Register 7
&20	Working Register 8
&24 to &FF	Acorn reserved

Table 18.2. Data arrangement in the SCCB.

This block is updated by the Level 1 handler. When the Gate On code is called the first four entries (ie, words 0 to 3) have been updated. Working R4 through to working R8 are for the user. In the example Voice Generator, working R4 is used as the wave table pointer. Working R5 is the absolute end of the data. Working R6 points to the absolute address of the LogAmpPtr. So these registers must be set up ready for the buffer fill routine.

Fill

This is called by the Level 1 handler when the DMA buffer needs to be filled with the next sample buffer. Again R9 points to the SCCB. The working registers must first be loaded as follows:

```
LDR  R4,[R9,#&10]   \ Waveform pointer
LDR  R5,[R9,#&14]   \ End of waveform
LDR  R6,[R9,#&18]   \ Amplitude scale table
```

The waveform data will be stored from R12 to R10 with an interleave value of R11. The interleave value is the way in which the sound channels are multiplexed. The following diagram may help to explain how the DMA buffer is filled with the waveform values from the different channels:

Voices	Bytes from Channels	Interleave value
1	1 1 1 1 1 1 1 1 b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 bA bB bC . . .	1
2	1 2 1 2 1 2 1 2 b0 b0 b1 b1 b2 b2 b3 b3 b4 b4 b5 b5 b6 . . .	2
4	1 2 3 4 1 2 3 4 b0 b0 b0 b0 b1 b1 b1 b1 b2 b2 b2 b2 b3 . . .	4
8	1 2 3 4 5 6 7 8 b0 b0 b0 b0 b0 b0 b0 b0 b1 b1 b1 b1 b1 . . .	8

It is a very simple matter to write the FOR ... NEXT loop in assembler to fill the buffer:

```
.Loop               \ Get wave form value into R0
STR R0,[R12],R11    \ Store the value and increment R11
CMP R12,R10         \ Reached end of the buffer?
BLT Loop
```

In the example R4 points to the waveform data. Therefore the next value in the waveform data must be loaded into R0 so it can be stored into the correct position in the DMA buffer, and R4 updated to point to the next byte.

When a piece of the waveform is played, every byte of the waveform will be placed in the DMA buffer. To double the pitch the data must be played twice as fast, in effect taking every other piece of waveform data. To half the pitch each piece of data must be placed in the buffer twice, eg,

Bytes stored in the buffer

Normal pitch	0 1 2 3 4 5 6 7 8 9 10 .....
Twice the pitch	0 2 4 6 8 10 12 14 16 18 20 .....
Half the pitch	0 0 1 1 2 2 3 3 4 4 5 .....

R2 contains the value used to advance the waveform and R4 points to the next piece of data. The following piece of assembler will advance the waveform, and load the value into R0:

```
ADD   R2,R2,R2,LSL #16
LDRB  R0,[R4,R2,LSR #24]
```

R2 is split up into two parts. Bits 16 to 31 are used as the phase accumulator, while bits 0 to 15 are used as the increment. The high byte of R2 is used to load the correct byte from the waveform. If, for example, every byte from the waveform was required, R2 would initially have the value &00000100:

Iteration	R2	High byte
0	&00000100	&00
1	&01000100	&01
2	&02000100	&02
3	&03000100	&03
...		

So to take every piece of data twice, R2 would have the value, &00000080:

Iteration	R2	High byte
0	&00000080	&00
1	&00800080	&00
2	&01000080	&01
3	&01800080	&01
4	&02000080	&02
...		

The value just obtained should be scaled to the current volume. On entry R1 contains the amplitude passed with the sound command, and this value also needs to be scaled. R6 points to the logarithmic amplitude lookup table through which the amplitude is scaled. The following assembler will perform this task:

```
AND   R1,R1,#&7F        \ Amplitude in range 0 to &7F
LDRB  R1,[R6,R1,LSL#1] \ Get value from table
MOV   R1,R1,LSR #1      \ Scale it
RSB   R1,R1,#127        \ make attenuation factor
```

This conversion is made as soon as the fill code is entered. The wave form values are also scaled and then stored in the DMA buffer:

```
SUBS  R0,R0,R1,LSL #1  \ Scale amplitude
MOVMI R0,#0            \ correct for underflow
STRB  R0,[R12],R11     \ Store value in DMA buffer
```

Once the buffer has been filled R4 is updated, so that the next time the buffer fill is called it points to the next byte in the waveform, this must then be stored back into the SCCB:

```
ADD  R4,R4,R2,LSR #24  \ Update the pointer
STR  R4,[R9,#&10]      \ store in SCCB
```

Gate Off

This is called when the waveform pointer reaches the end of the sampled data, or a new sound has been issued and the buffer needs to be flushed. Any remaining parts of the DMA buffer must then be filled with zeros, which is accomplished using the same FOR ... NEXT loop, but this time storing zeros in the DMA buffer:

```
MOV  R0,#0
.Floop
STRB R0,[R12],R11
STRB R0,[R12],R11
STRB R0,[R12],R11
STRB R0,[R12],R11
CMP  R12,R10
BLT  Floop
```

The Level 1 handler must now be informed that the sample has been finished, and it must flush the channel buffers next time. This is done by setting the control/status bits in the SCCB. The flags are as follows:

Bit	Meaning
7	Quiet, inactive
6	Kill pending
5	Initialise pending
4	Fill pending
3	Active
2	Overrun
1	Flush bit 1
0	Flush bit 0

The low two bits are used as a flush counter. If the fill is successful exit with R0 set to eight. To inform the Level 1 handler that the voice needs to be flushed, exit with R0 set to two or three.

If you stop sound abruptly then there may be an audible click. This can be avoided in two ways. Firstly alter the sampled data so that the sound 'dies' away. This is best done by extending the sample, and then adding a slight echo. The second, and preferable way, is to make the sound decay by entering a release phase, which may require buffer filling for a number of buffer periods. In this case close attention to the number of buffer fills (word 3 in the SCCB) is advised.

Update

Generally when a sound is played the previous sound is terminated, Gate Off, and then the new sound starts, Gate On. If however a smooth change over is required then setthe relevant bit in the amplitude parameter. The amplitude will be in the range &100 to &17F. However, if bit 7 is set, a smooth update will be requested. If this is the case then the update code is called, and the SCCB parameters are updated. In the example voice generator all this call does is to call the buffer fill code, as the fill code always refers to the SCCB for the amplitude and pitch.

Instantiate

Many channels may be attached to one voice, however complex voice generators may not be able to support more than one voice. Therefore, this code is called with the channel number–1 passed in R0. If the voice cannot support another channel then the value in R0 must be changed.

Free

This is called whenever a channel is detached from a voice, so that the Voice Generator can keep an up to date list of attached channels. This call *must* free the channel. The channel number-1 is passed in R0 and all registers must be preserved.

Install

This code is somewhat redundant as Voice Generators should be implemented as relocatable modules, and will therefore already be installed when called.

Voice Generator Code

The program listed opposite (listing 18.1) will convert a sampled waveform, created using the Armadillo sampling software, to a relocatable voice module. When run, you will be asked to supply two filenames – the sample filename and the filename for the module. Next enter the name of the voice, eg, crash or piano, and finally the channel number you wish to attach to the voice when it is loaded or initialised. Listing 18.2. will create a suitable data file that can be used with ModMaker (listing 18.1).

The sample will then be loaded, converted, the module header code added, and then saved to disc. The module will then automatically be loaded back

in. Typing *VOICES will list the new voice, while *MODULES and *HELP MODULES will supply information pertaining to the new voice.

You can quickly test the new module using a few simple star commands:

*FX 213 <pitch>	Sets the pitch of the bell
*FX 212 <volume>	Sets the bell volume (&80 loudest)
*FX 211 <channel>	Sets the bell channel number

The sample can be heard by pressing CTRL-G. It should not be too hard to find the correct playback pitch, and it is best to use the &100 upwards pitch values.

```
 10 REM >List18/1
 20 REM (c) Felix Andrew 1988
 30 REM Archimedes OS: A Dabhand Guide
 40 REM
 50 REM Voice Generator Module Creator
 60 :
 70 INPUT"Sample name ";name$
 80 INPUT"Module name ",mod$
 90 INPUT"Voice Title ";title$
100 :
110 X=OPENIN(name$):Sample_Size=EXT#X:CLOSE #X
120 PRINT"Voice ?";:V%=GET-48
130 DIM C% &1000+Sample_Size
140 :
150 FOR OPT%=4 TO 6 STEP2
160 P%=0:O%=C%:[OPT OPT%
170 .Head
180 EQUD 0
190 EQUD Init
200 EQUD Final
210 EQUD 0
220 :
230 EQUD Title
240 EQUD Help
250 EQUD 0
260
270 .Title EQUS title$+" Voice"+CHR$0:ALIGN
280 .Help  EQUS title$+CHR$9+"1.FA ("+MID$(TIME$,5,11)+")"+CHR$0
290
300 .Init   \Set the voice up
310 STMFD R13!,{R14}
320
330 ADR R0,VoiceBase:MOV R1,#0          \ Install voice in next
340 SWI "XSound_InstallVoice":BVS error \ free slot
350
360 STR R1,slot:MOV R0,#V%              \ Attaches channel V%
370 SWI "XSound_AttachVoice":BVS error  \ to new voice
380
390 LDMFD R13!,{PC}^
```

```
400
410 .slot EQUD 0
420
430 .Final                                  \ Remove the voice
440 STMFD R13!,{R14}
450
460 MOV R1,#0:MOV R0,#V%                    \ Detaches channel
470 SWI "XSound_AttachVoice":BVS error
480
490 LDR R1,slot                             \ Removes voice
500 SWI "XSound_RemoveVoice":BVS error      \ from slot
510
520 LDMFD R13!,{PC}^
530
540 .error
550 ADR R0,ers:LDMIA R13!,{PC}
560 .ers EQUS "A probelm has occured"+CHR$0:ALIGN
570
580 .VoiceBase              \ Sound Voice Control Block (SVCB)
590 B Fill                                  \ Fill code
600 B Update                                \ Update code
610 B GateOn                                \ Start code
620 B GateOff                               \ Release code
630 B Instance                              \ Instantiate
640 LDMFD R13!,{PC}                         \ Feature not supported
650 LDMFD R13!,{PC}                         \ Feature not supported
660 EQUD VoiceName-VoiceBase
670
680 .VoiceName
690 EQUS title$+STRING$(32-LEN name$,CHR$0)
700 ALIGN
710
720 .len EQUD Sample_Size
730
740 .Instance               \ any instance must use LogAmp table
750 STMFD R13!,{R0-R4}
760
770 MOV R0,#0
780 MOV R1,#0:MOV R2,#0
790 MOV R3,#0:MOV R4,#0
800 SWI "Sound_Configure"                   \ reads sound conf
810
820 LDR R0,[R3,#12]         \ Level 1 pointer to Log-scale table
830 ADR R1,LogAmpPtr
840 STR R0,[R1]
850
860 LDMFD R13!,{R0-R4,PC}
870
880 .LogAmpPtr EQUD 0
890
900 .Update
910 B Fill
920
```

```
 930 .GateOn
 940 ADR R4,Wave%                        \ Pointer-start of data
 950
 960 ADR R5,len:LDR R5,[R5]              \ Length of data
 970 ADD R5,R5,R4                        \ End of data
 980
 990 ADR R6,LogAmpPtr                    \ Location of LogAmp
1000 LDR R6,[R6]
1010
1020 STR R4,[R9,#&10]                    \ Store these
1030 STR R5,[R9,#&14]                    \ values in the
1040 STR R6,[R9,#&18]                    \ working registers
1050
1060 .Fill
1070                                     \ Passed
1080                                     \ R9 Points to SCCB
1090
1100 LDR R1,[R9,#&00]                    \ Get volume
1110 LDR R4,[R9,#&10]                    \ Pointer to data
1120 LDR R5,[R9,#&14]                    \ End address of data
1130 LDR R6,[R9,#&18]                    \ Location of LogAmp
1140 LDR R7,[R9,#&04]                    \ pitch
1150
1160 \ SCCB Format
1170 \ 00     Amplitude
1180 \ 04     Pitch
1190 \ 08
1200 \ 0C
1210 \ 10     Pointer to data
1220 \ 14     End of data
1230 \ 18     Location of LogAmp
1240
1250 AND    R1,R1,#&7F                   \ Amplitude 0 to 127
1260 LDRB   R1,[R6,R1,LSL #1]            \ get scaled volume
1270 MOV    R1,R1,LSR #1
1280 RSB    R1,R1,#127                   \ attenuation factor
1290
1300 .FillLoop
1310 CMP R4,R5
1320 STRGE R4,[R9,#&10]                  \ Update pointer SCCB
1330 BGE GateOff                         \ No more sample data
1340 ]:FOR L=0 TO 3:[OPT OPT%
1350 ADD   R7,R7,R7,LSL #16              \ Advance block pointer
1360 LDRB  R0,[R4,R7,LSR #24]            \ Get data
1370 SUBS  R0,R0,R1,LSL #1               \ Scale data
1380 MOVMI R0,#0
1390 STRB  R0,[R12],R11                  \ Store data in DMA buf
1400 ]:NEXT:[OPT OPT%
1410
1420 ADD   R4,R4,R7,LSR #24              \ Update pointer
1430 BIC   R7,R7,#&FF000000
1440
1450 CMP   R12,R10                       \ Fill DMA buffer
```

```
1460 BLT   FillLoop
1470
1480 STR R4,[R9,#&10]                    \ Update pointer in SCCB
1490
1500 CMP   R4,R5                         \ Are we at the end ?
1510 MOVLT R0,#8                         \ Voice remains active
1520 MOVGE R0,#2                         \ Flush voice next time
1530 LDMFD R13!,{PC}                     \ Back to Level 1 handler
1540
1550 .GateOff                            \ Flush DMA buffer
1560 MOV R0,#0
1570 .FlushLoop
1580 STRB  R0,[R12],R11
1590 STRB  R0,[R12],R11
1600 STRB  R0,[R12],R11
1610 STRB  R0,[R12],R11
1620 CMP R12,R10
1630 BCC FlushLoop                     \ Store 0s in DMA buffer
1640 MOV R0,#2                         \ Flag to be flushed again
1650 LDMFD R13!,{PC}
1660
1670 .Wave%                            \ Wave table information
1680 ]:P%+=Sample_Size:[OPT OPT%
1690 .end
1700 ]:NEXT
1710 D%=P%
1720 PRINT"Start convert ";
1730 OSCLI "LOAD "+name$+" "+STR$~(Wave%+C%)
1740 PRINT" converting ";
1750 PROCconc:CALL convert
1760 PRINT
1770 OSCLI "SAVE   "+mod$+" "+STR$~C%+"+"+STR$~D%
1780 OSCLI "SETT.  "+mod$+" FFA"
1790 OSCLI "RMLOAD "+mod$
1800 END
1810
1820 DEF PROCconc
1830 DIM CO% &200:FOR OPT%=0 TO 2 STEP2:P%=CO%:[OPT OPT%
1840 .destp EQUD Wave%+C%
1850 .amm EQUD Sample_Size
1860
1870 .log_tab
1880 EQUS STRING$(255,"F")
1890 EQUD 00
1900 ALIGN
1910
1920 .convert
1930 STMFD R13!,{R0-R3,R14}
1940 MOV R3,#0
1950 ADR R2,log_tab
1960 .logcl
1970  SUB R0,R3,#&80
1980  MOV R0,R0,LSL #24
```

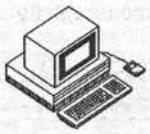
```
1990  SWI "Sound_SoundLog"
2000  STRB R0,[R2,R3]
2010  ADD R3,R3,#1
2020  CMP R3,#&100        \ Generate default 256 entry log table
2030  BCC logcl
2040  LDR R1,destp
2050  LDR R2,amm
2060  ADR R3,log_tab
2070  .cl
2080  LDRB R0,[R1]        \ Converts sample data by using log table
2090  LDRB R0,[R3,R0]
2100  STRB R0,[R1],#1
2110  SUBS R2,R2,#1
2120  BPL cl
2130  LDMFD R13!,{R0-R3,PC}
2140  ]:NEXT
2150  ENDPROC
```

Listing 18.1. Voice Generator Module Creator.

```
  10  REM >List18/2
  20  REM (c) Felix Andrew 1988
  30  REM Archimedes OS: A Dabhand Guide
  40  REM
  50  REM Creates a simple wavetable for use
  60  REM with ModMaker
  70  :
  80  DIM data% 1024*10
  90  DIM S(360)
 100  FOR T%=0 TO 360
 110   S(T%)=SIN(RAD(T%))*230/5
 120  NEXT
 122  MODE 8
 130  ORIGIN 0,500
 140  MOVE 0,-&80*2:PLOT 1,1280,0
 150  MOVE 0, &80*2:PLOT 1,1280,0
 152  PRINTTAB(3,3)"Generating a waveform"
 160  FOR X%=0 TO 1024*10
 170   P1=(X%)MOD 360
 180   P2=(X%*2)MOD 360
 190   P3=(X%*10)MOD 360
 200   A=SIN RAD(X% DIV 57)
 210   data%?X%=((S(P2)+S(P1)+S(P3))*A)+&80
 220   PLOT 69,X%/8,((data%?X%)-&80)*2
 230  NEXT
 240  OSCLI "SAVE Data "+STR$~data%+"+"+STR$~(1024*10)
```

Listing 18.2. Sound Sample.

19 : Character Input/Output



Simple Input/Output

Some of the most fundamental operations provided by the Operating System are those which allow it to get characters typed by the user and to print characters on the display for the user to read. These functions are essential to allow the user to communicate with the OS and they are also a vital component of all applications software.

The OS provides a sophisticated set of SWIs which allows single-character communication between the computer and the user through its display and keyboard. Many of these routines are provided for compatibility with the BBC MOS; others are new or improved facilities.

This section is devoted to the mechanisms for character input/output (I/O) – it describes the basis upon which the OS's character I/O is built and examines the most useful of the many SWIs that the OS provides.

Character Input

In communicating with the Operating System, and indeed with applications programs, we need to be able to deal with anything from a single keypress to multiple lines of text. The fundamental aspect of both of these extremes is the character – a single unit which may be a letter of the alphabet, a punctuation symbol, a control code and so on. Sequences of characters are known to the OS as 'streams' (think of the 'flowing' of characters) and the OS allows one of three possible input streams to be active at any moment.

The three possible character input streams are:

- the keyboard itself
- the RS423 port input channel
- a *EXEC file

These streams are mutually exclusive: only one of them may be active at any moment (because this is a single-user, single-tasking OS at present). So, the most fundamental character input operation is to allow us to decide

where characters are coming from. Exactly this mechanism is provided by one of OS_Byte SWIs – the compatible equivalent of *FX 2 under the BBC MOS.

OS_Byte (SWI &06) Function &02 Select Input Stream

This SWI takes a parameter in R1 which selects between the keyboard and the RS423 port as follows:

Value	Source
0	Keyboard – with RS423 disabled
1	RS423
2	Keyboard – with RS423 enabled

It also returns the previous selection in R1 as either 0 or 1 (again relating to the table above). The difference between the two keyboard source types is that, when the RS423 port is enabled, characters will be received and buffered by the RS423 port even though it is not selected as the current input stream.

Getting Hold of Single Characters

Having selected which source of characters is in use, we actually need to fetch the characters. Once again, BBC MOS compatibility obliges us to obey certain rules. In particular, a SWI is provided called 'OS_ReadC' which behaves in the same way as its BBC MOS counterpart (OSRDCH). OS_ReadC waits for a character to be entered and then returns with its ASCII code.

OS_ReadC (SWI &04) Wait for and Return an ASCII Character

No calling parameters are required. The SWI returns with R0 containing the ASCII code of the character and the Carry flag 'C' clear if it is valid. If an ESCAPE condition occurs (eg, if the ESCAPE key was pressed) then the carry flag is set and R0 will contain the ASCII escape code &1B.

Alternatively, we may check to see if a key has been pressed without waiting for one. This is provided by the BBC MOS compatible call OS_Byte &81 which reads a key within a time limit (which may be zero).

OS_Byte (SWI &06) Function &81 Return ASCII Key with Time Limit

On entry, this SWI requires R1 and R2 to contain a time limit (in centiseconds) with the low byte in R1 and the high byte in R2 (this arrangement is for BBC MOS compatibility). When the call returns, R1 contains the ASCII code of the key pressed, or &FF if no key was pressed within the specified time. Also, R2 will contain a flag indicating the result, which will be zero if a valid key was pressed, &1B if an ESCAPE condition occurred or &FF if a timeout occurred.

OS_Byte &81, when supplied with a negative time limit, can also perform a specific key scan function as it does under the BBC MOS. This is described later.

Whole Lines of Characters

Because it is so common for a program to need to input a whole line of characters at a time, a standard routine is provided to do so. In fact, two routines are provided, one of which is BBC MOS compatible. It is preferable that new software uses the latter, since the lifetime of the former is likely to be limited.

The effect of these routines is to accept a sequence of characters from the current input stream and store them in the input buffer. The Delete character is dealt with in the usual way, viz. it is copied to the display and removes the last character from the buffer. CTRL-U is also decoded to delete the entire contents from the buffer and the displayed line accordingly. The line may be terminated by entering RETURN, ENTER, CTRL-J or by pressing the ESCAPE key. The line returned in the buffer always has a final ASCII 13 appended to it, regardless of the way in which input was actually terminated.

The BBC MOS compatible form is the familiar OSWORD &00 call – it will read a line of text entered by the user into a buffer. A number of frills are provided too: we may limit the maximum and minimum ASCII codes of characters placed in the buffer (to prevent control codes being included, for example) and we may limit the overall length of the line (to prevent our buffer overflowing). The OS_Word form of this routine is detailed opposite.

OS_Word (SWI &07) Function &00
Read a Line from the Input Stream

On entry, this call needs to be provided with a parameter block of bytes. The address of this block should be placed in R1 and should be arranged as follows:

Position	Contents
R1+0	LSB of buffer address
R1+1	MSB of buffer address
R1+2	Maximum number of characters
R1+3	Lowest permissible ASCII code
R1+4	Highest permissible ASCII code

When the call returns, R2 will contain the length of the line entered (not including the terminating character), and the Carry flag 'C' will be clear normally and set if an ESCAPE condition occurred during input. The OS form of this routine is SWI OS_ReadLine.

OS_ReadLine (SWI &0E)
Read a Line from the Input Stream

On entry, the same parameters must be supplied as for the OS_Word call, but they are packed into the registers slightly differently:

Register	Contents
R0	Pointer to buffer
R1	Maximum number of characters
R2	Lowest permissible ASCII code
R3	Highest permissible ASCII code

On return, R1 will contain the length of the line entered, and the Carry flag will normally be clear but set if an ESCAPE condition occurred.

Keyboard Control Functions

As well as the character and line input functions, a large number of keyboard control calls are also provided to allow low-level modification of the way the keyboard operates. Almost all of these calls are provided through the OS_Byte SWI and are, therefore, compatible with the BBC MOS. Table 19.1 describes the calls supported and mentions where differences between the BBC MOS and the OS exist.

OS_Byte	Function
&04	Cursor key functions
&ED	Read/write cursor key function
&0B	Write keyboard auto-repeat delay
&C4	Read/write keyboard auto-repeat delay
&0C	Write keyboard auto-repeat rate
&C5	Read/write keyboard auto-repeat rate
&76	Force keyboard LEDs to correspond to flags
&78	Simulate key depression
&79	Keyboard scan
&7A	Keyboard scan from 16 (not for SHIFT, CTRL, ALT or the mouse)
&81	Keyboard scan for specific key
&7C	Clear ESCAPE condition
&7D	Set ESCAPE condition
&7E	Acknowledge ESCAPE condition
&B2	Read/write keyboard semaphore
&C8	Read/write BREAK and ESCAPE controls
&C9	Read/write keyboard disable flag
&CA	Read/write keyboard status byte
&D8	Read/write function key length
&DB	Read/write TAB key character code
&DC	Read/write ESCAPE character code
&DD	Read/write translation of codes &C0-&CF
&DE	Read/write translation of codes &D0-&DF
&DF	Read/write translation of codes &E0-&EF
&E0	Read/write translation of codes &F0-&FF

Table 19.1. OS_Byte SWI calls.

The keyboard status byte contains a bit pattern which represents the perceived state of the various shift keys for the OS. These bits are arranged as follows:

Bit	Meaning When Set
0	Incomplete ALT
1	SCROLL LOCK on
2	NUM LOCK off
3	SHIFT on
4	CAPS LOCK off
5	Always 1
6	CTRL on
7	SHIFT CAPS on

These four calls allow the interpretation of the range of codes from &C0 to &FF to be modified. The parameter in R1 is interpreted as follows:

Value	Meaning
0	Ignore these codes
1	Expand to appropriate function key string
2	Precede the code with a null (0)
3-255	Generate (code MOD 16)+value in R1

These codes can usually only be generated by receiving them through the RS423 input channel or inserting them into the keyboard buffer. However, four of the function keys can generate codes in this region:

Key	Code	+SHIFT	+CTRL	+SHIFT+CTRL
f10	&CA	&DA	&EA	&FA
f11	&CB	&DB	&EB	&FB
f12	&CC	&DC	&EC	&FC
Insert	&CD	&DD	&ED	&FD

Note that the code &CA for function key f10 is inserted into the keyboard buffer on reset to allow it to simulate the *KEY 10 function under BBC MOS.

&E1	Read/write function key interpretation
&E2	Read/write SHIFT+function key interpretation
&E3	Read/write CTRL+function key interpretation
&E4	Read/write SHIFT+CTRL+function key interpretation

These four calls control the interpretation of codes in the range &80-&BF, within which the main function keys, cursor keys and so forth appear. The parameter supplied in R1 affects how the codes are treated:

Value	Meaning
0	Ignore these codes
1	Expand to appropriate function key string
2	Generate a null (0) followed by code
3-255	Generate (code MOD 16)+value in R1

The fundamental codes produced by the function keys are summarised below:

Key	Code
Print	&80
f1	&81
f2	&82
f3	&83
f4	&84

Key	Code
f5	&85
f6	&86
f7	&87
f8	&88
f9	&89
COPY	&8B
Left-arrow	&8C
Right-arrow	&8D
Down-arrow	&8E
Up-arrow	&8F

Where one or both of the SHIFT and CTRL keys is pressed, a multiple of 16 is added to the codes, so SHIFT+PRINT produces &90, CTRL+PRINT produces &A0 and SHIFT+CTRL+PRINT produces &B0.

The purpose of R1=2 in the previous table is to allow ISO standard foreign characters to be dealt with correctly and processed separately from function keys. When permitting the entry of ISO characters, it is prudent to set the function keys to this mode in order that they may be distinguished from foreign characters with the same code.

&E5	Read/write ESCAPE key status
&E6	Read/write ESCAPE effect
&EE	Read/write numeric keypad code layout

Note that the codes generated by the numeric keypad vary according to whether the NUM LOCK key is lit or not – consult the Programmer's Reference Manual for more information.

Character Output

Just as there are several character input streams, so there are several output streams. The four main output streams are:

- The VDU stream, which drives the display
- The RS423 port output channel
- The printer stream
- The *SPOOL file (when activated)

The VDU stream accepts sequences of characters and either displays them, or uses them to control character output. Besides the many text and graphics functions the VDU drivers provide, they also allow control over whether or not characters are sent to the printer stream. The various 'VDU

codes' which control these effects will be familiar to users of the BBC MOS and are documented extensively in the Archimedes User Guide.

The RS423 port output channel can be used to output characters in one of two ways: it may either operate as an adjunct to the VDU stream, or it may be driven by the printer stream if a serial printer is to be connected. These two options are discussed below.

Finally, the spool file makes a copy of whatever is sent to the VDU stream in a specified file. This allows sequences of VDU codes to be 'recorded' and subsequently 'replayed' using *PRINT (which takes bytes from the file and sends them directly to the VDU drivers).

Selecting Which Output Streams are to be Used

A most important output-related call is the one which selects which output stream(s) are to be used. Of course, unlike the input streams, more than one output stream may be selected at once so, for example, we may see results both displayed and printed. The selection of output streams is effected by a number of controls, the most important being the BBC MOS compatible call OS_Byte &03.

OS_Byte (SWI &06) Function &03 Select Output Streams

On entry, the call requires the bottom byte of R1 to contain a pattern of bits to select the required combination of output streams. These bits are interpreted as follows:

Bit	Meaning if Set
0	RS423 output enabled
1	VDU stream disabled
2	VDU printer driver disabled
3	Non-VDU printer driver enabled
4	Spooling disabled
5	Use VDUXV instead of VDU drivers
6	Printer disabled except for VDU 1,n
7	Not used

The default setting of these bits is all zero, meaning that the VDU drivers, the VDU printer drivers and spooling are all enabled. The exact interpretation of these various bits is described in the relevant parts of the rest of this section.

Another OS_Byte Call – function &EC – may be used to read/write this setting in the normal BBC MOS way.

Selecting the VDU Stream

The VDU stream is the fundamental route through which most characters are sent. The VDU driver software accepts characters from the VDU stream and either displays them (in the case of printable characters), acts upon them (in the case of control codes) and/or sends them on to the printer (when printing has been enabled).

When bit one of the output select byte is set, the VDU stream is completely disabled, preventing any characters from being displayed or being sent to the printer.

When bit five is set, the OS calls to the VDU drivers are replaced by calls through the VDU extension vector (VDUXV) to allow replacement VDU drivers to be installed. See the chapter on vectors (Chapter 20) for more information.

Selecting the RS423 Output Stream

Characters may be sent to the buffered RS423 output stream in one of two ways:

1. When bit 0 of the output select byte is set, characters sent to the VDU drivers are duplicated into the RS423 output buffer and then transmitted under interrupt control of the RS423 hardware in the usual way.
2. Alternatively, an RS423 printer may be selected using the printer selection call (see later) in which case control of whether characters are sent to the output channel is handed over to the printer controls discussed below.

Selecting the Printer Stream

Usually, characters to be sent to the printer pass through the VDU drivers. Two control codes are used to tell the VDU drivers whether to pass on characters to the printer: one of them turning the printer stream on and the other turning it off. However, the output select bits have the following overriding effects:

- When bit two is set, no characters will be sent to the printer at all.

- When bit six is set, characters may only be sent to the printer using the VDU 1,n sequence (this requires the VDU drivers to be enabled).
- Finally, when bit three is set, all characters sent to the VDU drivers are duplicated to the printer – regardless of whether the appropriate codes have been sent to the VDU drivers to enable printing.

Several different kinds of printer may be attached to Archimedes computers; the call OS_Byte &05 allows one of the alternatives to be chosen.

OS_Byte (SWI &06) Function &05 Select Printer Driver

On entry, this call takes a parameter in R1 which indicates the chosen printer driver. The possible values are shown below:

Value	Printer Driver Selected
0	Printer sink (characters discarded)
1	Parallel (Centronics) printer
2	RS423 output channel printer
3	User printer
4	Econet network printer
5-255	Other user printers

The previous setting is placed in R1 on return from this call. It corresponds exactly to its BBC MOS equivalent, as does the associated OS_Byte &F5 which allows the current setting to be read.

The following OS_Bytes may also be used in a BBC MOS compatible way:

&06	Write printer ignore character
&F6	Read/write printer ignore character
&B6	Read/write NOIGNORE status – bit 7 set for 'no ignore' character

Selecting the Spool File Stream

Initially, spooling of printed output must be selected by issuing a *SPOOL command to open the spool file. Once this has been done, characters subsequently sent to the VDU drivers will be duplicated into the selected spool file until spooling is turned off again.

However, bit four of the output select byte can be set to temporarily disable spooling without actually closing the file etc. This may be useful if you do not want graphics sequences sent to the spool file, for example.

An OS_Byte call is provided to read the file handle of the current spool file, allowing you to take control of the spooling process at a lower level, if desired.

OS_Byte (SWI &06) Function &C7 Read/Write Spool File Handle

On entry R1 and R2 should be set as usual for this type of call, viz:

To read the current handle:

R1=0, R2=255

with the result returned in R1.

To write the handle:

R1=new handle (supplied by OS_Find), R2=0

with the new setting returned in R1.

Character Output to the Selected Streams

Having established how to select the various output streams, we now need to be able to actually send characters to the selected stream(s). The most fundamental call to achieve this is the BBC MOS compatible call OS_WriteC, which is equivalent to the BBC MOS call OSWRCH.

OS_WriteC (SWI &00) Write a Character to the Output Streams

On entry the character should be in the bottom byte of R0. The character is sent to all of the currently selected output streams. No result is returned.

```
120 MOV R0,#ASC"A"
130 SWI "OS_WriteC"
```

There is also a range of SWI numbers, which have the same effect as calling OS_WriteC but display an ASCII character whose number is determined from the SWI number. The range of SWI numbers runs from &100 to &1FF, with the ASCII character whose code is the SWI number less &100 being printed, ie, SWI &120 displays a space and so forth.

OS_WriteI (SWIs &100 to &1FF)
Write Specified ASCII Character

This range of SWIs takes no entry parameters and returns no results. An example of these SWIs is provided at the end of the chapter (listing 19.1). This call is invaluable for debugging, since it allows the programmer to output a character when the execution of the program reaches the SWI instruction without any of the registers being corrupted.

Additionally, a number of calls are provided to output multiple character 'strings'. These strings may be in one of two forms: either immediately following the SWI instruction (an 'in-line' string) and terminated by a zero, or at an arbitrary location in memory, again terminated by a zero. It is crucial, in both cases, to ensure that the string begins on a byte boundary, so it is wise to use the ALIGN directive in the ARM BASIC assembler where appropriate.

OS_WriteS (SWI &01)
Write an 'In-line' String to the Output Streams

The call takes no entry parameters because the string to be written begins immediately after the instruction. It must be terminated by a zero. Execution of instruction begins at the next word after the end of the string.

```
160 SWI "OS_WriteS"
170 EQUS "This is a test"
180 EQUB 0
190 ALIGN
```

There is a slight restriction to be wary of when using OS_WriteS. It is possibly to make any ARM instruction execute conditionally by suffixing various condition codes to it. It may be thought, therefore, that we could write code like the following.

```
10  SWI "OS_ReadC"
20  CMP R0,#ASC"y"
30  SWIEQ "OS_WriteS"
40  EQUS "Option selected"
50  EQUB 0
60  ALIGN
70
80  < rest of program >
```

The idea is for the OS_WriteS instruction to write "Option Selected" *only* when a 'y' has been previously entered.

It is certainly true that if a 'y' is not entered, the SWI instruction will not execute. However, in this case, the ARM program counter will not be manipulated by the SWI to start execution after the string which follows it. The result of this is that the ARM will continue execution immediately after the SWI "OS_WriteS" instruction itself. It will attempt to interpret the characters in the string as ARM instructions and execute them! This can cause some spectacular crashes and the cause of the failure is often hard to track down.

In practice, therefore, a conditional execution suffix should never be added to a SWI "OS_WriteS" instruction.

OS_Write0 (SWI &02)
Write an Indirect String to the Output Streams

On entry, R0 must point to the start of the string. The string must be terminated by a zero.

```
220 ADR R0,a_string
230 SWI "OS_Write0"
```

Where the string is of fixed length an alternative form may be used: OS_WriteN (SWI &46) – write a fixed length string to the output streams. On entry, R0 must point to the start of the string and R1 must contain the number of bytes to write (not necessarily the same as the length of the string).

```
220 FOR i%=1 TO LEN a$
230 SYS "OS_WriteN",a$,i%
240 SYS "OS_NewLine"
250 NEXT
```

Because text printing is such a common activity, the OS contains a primitive text formatting call which deals with common control codes: this is the SWI OS_PrettyPrint. OS_PrettyPrint takes a string and prints out each character, translating any embedded carriage returns (ASCII 13), tabs (ASCII 9) and 'hard spaces' (code 31). The Archimedes usually treats carriage returns as simply meaning 'move back to the start of the line' without moving the printing position down one line as, for example, the BBC MOS did. Tabs are expanded to align text to the start of the next eight-character column, and text which overflows a display line is 'word-wrapped' at the next space, unless it is a 'hard' space in which case it is left alone. The combination of these translation facilities is useful, if a little simplistic.

OS_PrettyPrint (SWI &44)
Format and Print an ASCII String

On entry, R0 must point to the start of a zero-terminated string. The text of the string is translated, where necessary, and all characters sent to OS_WriteC. No results are returned.

The last useful OS output SWI simply prints a carriage return (ASCII 13) followed by a line feed (ASCII 10), so it is aptly named OS_NewLine.

OS_NewLine (SWI &03)
Move Print Position to Start of Next Line

The call takes no parameters and returns no results.

Listings

```
 10  REM >List19/1
 20  REM by Nicholas van Someren
 30  REM Archimedes OS: A Dabhand Guide
 40  REM (c) Copyright AvS and NvS 1988
 50  :
 60  DIM buffer 100,code% 2000
 70  FOR pass%=0 TO 3 STEP 3
 80  P%=code%
 90  [OPT pass%
100  .start
110  .swi0
120  MOV R0,#ASC"A"
130  SWI "OS_WriteC"
140  :
150  .swi1
160  SWI "OS_WriteS"
170  EQUS "This is a test"
180  EQUB 0
190  ALIGN
200  :
210  .swi2
220  ADR R0,a_string
230  SWI "OS_Write0"
240  :
250  .swi3
260  SWI "OS_NewLine"
270  :
280  .swi4
290  SWI "OS_WriteS"
300  EQUS "Press a key..."
310  EQUB 0
320  SWI "OS_ReadC"
```

```
330  SWI "OS_WriteC"
340  SWI "OS_NewLine"
350  :
360  .swi5
370  ADR R0,a_command_string
380  SWI "OS_CLI"
390  :
400  .swiE
410  SWI "OS_WriteS"
420  EQUS "Enter a string, followed by Return."
430  EQUB 0
440  ALIGN
450  ADR R0,buffer
460  MOV R1,#100
470  MOV R2,#32
480  MOV R3,#126
490  SWI "OS_ReadLine"
500  ADD R4,R0,R1
510  MOV R5,#0
520  STR R5,[R4]
530  SWI "OS_Write0"
540  SWI "OS_NewLine"
550  :
560  .swi21
570  MOV R0,#16
580  ADR R1,a_number_string
590  SWI "OS_ReadUnsigned"
600  :
610  .swi23
620  ADR R1,buffer
630  MOV R3,#0
640  .swi23loop
650  ADR R0,a_variable_name
660  MOV R2,#100
670  MOV R4,#3
680  SWI "XOS_ReadVarVal"
690  MOVVSS PC,R14
700  MOV R0,R3
710  SWI "OS_Write0"
720  SWI "OS_NewLine"
730  B swi23loop
740  :
750  .a_string
760  EQUS "This is a silly little string"
770  EQUB 0
780  ALIGN
790  :
800  .a_command_string
810  EQUS "EX $"
820  EQUB 0
830  ALIGN
840  :
850  .a_number_string
```

```
860  EQUS "&4DE23"
870  EQUB 0
880  ALIGN
890  :
900  .a_variable_name
910  EQUS "Alias$*"
920  EQUB 0
930  ALIGN
940  ]:NEXT pass%
950  PRINT'"Press Return to test.."
960  REPEAT UNTIL GET=13
970  CLS
980  CALL start
990  END
```

Listing 19.1. Simple I/O.

```
 10  REM >List19/2
 20  REM by Nicholas van Someren
 30  REM Archimedes OS: A Dabhand Guide
 40  REM (c) Copyright AvS and NvS 1988
 50  :
 60  REM Take a string and for each of its characters,
 70  REM call SWI &100 plus the value of the character.
 80  :
 90  a$="This is a test"+CHR$13+CHR$10
100  FOR char%=1 TO LEN(a$)
110  SYS &100+ASC(MID$(a$,char%,1))
120  NEXT
130  END
```

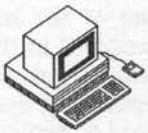
Listing 19.2. WriteI example.

```
 10  REM >List19/3
 20  REM by Nicholas van Someren
 30  REM Archimedes OS: A Dabhand Guide
 40  REM (c) Copyright AvS and NvS 1988
 50  :
 60  MODE 0
 70  :
 80  REM Make a$ a tabulated string and print it 'prettily'.
 90  :
100  a$="THIS"+CHR$9+"IS"+CHR$9+"TEST"+CHR$13+"and"+CHR$9+
"so"+CHR$9+"is"+CHR$9+" this."
110  SYS "OS_PrettyPrint",a$
120  PRINT
130  :
140  REM Read data and plot it.
```

```
150  :
160  REPEAT
170  READ m%,x%,y%
180  SYS "OS_Plot",ABS(m%),x%,y%
190  UNTIL m%<0
200  :
210  REM Print out substrings of a$
220  FOR i%=1 TO LEN a$
230  SYS "OS_WriteN",a$,i%
240  SYS "OS_NewLine"
250  NEXT
260  END
270  :
280  DATA 4,500,500
290  DATA 5,700,500
300  DATA 5,700,700
310  DATA -5,500,500
```

Listing 19.3. PrettyPlot.

20 : Vectors



Just as the BBC MOS made extensive use of vectors, so the Archimedes OS does too – allowing OS routines to be bypassed, if necessary. The mechanism for using vectors has been rationalised so that a consistent view of vector handling is provided to all software.

Before looking at the details of the software vectors provided by the Operating System, it is worth noting that the ARM itself also uses vectors. These are completely distinct from the Operating System ones and are called the hardware vectors.

The Hardware Vectors

The hardware vectors are a series of locations fixed in memory which the ARM will jump to if it encounters a situation which it cannot itself directly deal with. For example, the user may be attempting to execute an unknown instruction or to access some non-existent memory location. In these and other similar situations, the ARM breaks off its current task and begins execution at the appropriate hardware vector. When the system wasinitialised, the OS will have placed branch instructions in each of these vectors so that the ARM will jump to a suitable piece of code within the Operating System to handle the situation which is causing the problem.

The various hardware vectors are listed below in table 20.1:

Vector Location	Called in response to
&00000000	Machine RESET
&00000004	Unknown Instruction
&00000008	Software Interrupt (SWI)
&0000000C	Pre-fetch Abort
&00000010	Data Abort
&00000014	Address Exception
&00000018	Interrupt occurring (IRQ)
&0000001C	Fast Interrupt occurring (FIRQ)

Table 20.1. The Hardware vectors.

It is theoretically possible for users to place their own branch instruction in one of the hardware vectors. This would allow some user code to be executed in the corresponding circumstance. However, this is a very risky occupation, and there is usually an easier to use facility provided by the Operating System. For example, to add extra SWI instructions we do not need to replace the SWI hardware vectors as the OS provides an 'Unknown SWI' software vector.

One common reason for manipulating the hardware vectors is to change the machine's response to memory access faults. If some non-existent memory is accessed then one of the memory fault vectors, &0000000C to &00000014, is called. The normal effect of this is for the Operating System to report a fatal error and stop executing the current task.

In some cases, for example when writing a memory editor, this is not a very desirable thing to happen. It would be better to simply warn the user that a particular location is invalid and allow editing to continue of the rest of memory.

The subroutine presented below can do exactly this. On entry to the routine, R0 should contain the address of the memory location to be read. On exit, R0 will contain the data byte found at this location or –1 if the access to the memory location caused a fault. Note that, the original contents of the hardware vectors are preserved on entry and restored on exit. This is vital, so that a real memory fault at some later time does not cause the subroutine to be suddenly re-entered!

```
 10 REM >List20/1
 20 REM Archimedes OS: A Dabhand Guide
 30 REM (c) Mike Ginns 1988
 40 REM Example of manipulating the hardware vectors
 50 REM
 60 REM
 70 :
 80 DIM protected_memory_read 1024
 90 FOR pass = 0 TO 3 STEP 3
100 P% = protected_memory_read
110 [
120 OPT pass
130 :
140 \ On entry R0 = address of byte to read
150 \ On exit R0 = contents of location
160 \ OR R0 = -1 if memory address invalid
170 :
180 \ Enter SVC mode. May not be able to access hardware vectors
190 \ from user mode in future operating systems
200 :
210 SWI "OS_EnterOS"
220 :
```

```
230 MOV R3,#0
240 LDR R1,[R3,#&10]   \ Old contents of data abort vector
250 LDR R2,[R3,#&14]   \ Old contents of exception vector
260 STMFD R13!,{R1,R2} \ Preserve previous contents on stack
270 :
280 :
290 \ Instruction pattern for 'ANDNV R0,R0,R0' (NOP)
300 :
310 MOV R1,#&F0000000
320 STR R1,[R3,#&10]   \ Place 'NOP' in data abort vector
330 :
340 :
350 \ Construct suitable branch instruction to jump to the
360 \ memory  fault code from the exception vector
370 :
380 ADR R1,memory_fault    \ Routine called if illegal access
390 SUB R1,R1,#20+8         \ Vector at &14 + 8 for PipeLine
400 MOV R1,R1,LSR#2         \ Should be a word address
410 ORR R1,R1,#%11101010<<24   \ Include branch op with address
420 :
430 STR R1,[R3,#&14]        \ Store branch in exception vector
440 :
450 :
460 \ Try to read the byte of memory, the address of which is
470 \ in register R0. If this results in a memory fault,
480 \ then the 'memory_fault' code is executed.
490 :
500 LDRB R0,[R0]                \ Access memory location
510 .after_load_instruction     \ Reached even if memory fault
520 :
530 LDMFD R13!,{R1,R2}          \ Get original vectors from stack
540 MOV R3,#0
550 STR R1,[R3,#&10]            \ Restore data abort vector
560 STR R2,[R3,#&14]            \ Restore exception vector
570 :
580 TEQP PC,#0                  \ Back to user mode
590 MOVNV R0,R0
600 :
610 MOV PC,R14                  \ Return to caller
620 :
630 :
640 :
650 \ This code is only executed if a memory fault occured when
660 \ the specified location was accessed. It places -1 in R0
670 \ then jumps back to the instruction immediately after the
680 \ LDR instruction.
690 :
700 .memory_fault
710 MVN R0,#0  \ Move -1 into R0
720 B after_load_instruction
730 :
740 ]
750 NEXT
```

```
760  :
770  :
780  REPEAT
790  :
800  PRINT'
810  INPUT "Enter address of byte to be examined : " addr$
820  PRINT
830  A% = EVAL(addr$)
840  result = USR( protected_memory_read )
850  :
860  IF result = -1 THEN
870  PRINT "Address invalid"
880  ELSE
890  PRINT "Data at this location is : " ;result
900  ENDIF
910  :
920  UNTIL FALSE
```

Listing 20.1. Manipulating the hardware vectors.

The Operating System Software Vectors

The OS software vectors may be intercepted much more easily and safely than the hardware ones and should be used in preference wherever possible. A number of SWIs are provided to help in manipulating the vectors. The most important are OS_Claim and OS_Release.

The procedure for using vectors is as follows. A call to OS_Claim will attach your code to the front of the chain of claimants for that particular vector. You may then either, deal with a call to the vector and pass it on, or 'hold' the vector and prevent other claimants from being called. The latter situation must obviously be used with considerable care, since holding an important vector such as one of the interrupt vectors or the error handler's vector is likely to cause trouble.

The vectors that the OS uses are summarised in table 20.2 below, together with the SWIs and operations which are indirected through them.

Name	Number	Function
ErrorV	&01	OS_GenerateError
IrqV	&02	Normal interrupt vector
WriteCV	&03	OS_WriteC
ReadCV	&04	OS_ReadC
CliV	&05	OS_CLI
ByteV	&06	OS_Byte
WordV	&07	OS_Word

Name	Number	Function
FileV	&08	OS_File
ArgsV	&09	OS_Args
BGetV	&0A	OS_BGet
BPutV	&0B	OS_BPut
GBPBV	&0C	OS_GBPB
FindV	&0D	OS_Find
ReadLineV	&0E	OS_ReadLine
FSControlV	&0F	OS_FSControl
EventV	&10	Event vector
InsV	&14	Buffer insertion
RemV	&15	Buffer removal
CnpV	&16	Buffer count/purge
UKVDU23V	&17	Unknown VDU 23 code
UKSWIV	&18	Unknown SWI
UKPLOTV	&19	Unknown VDU 25 code
MouseV	&1A	OS_Mouse
VDUXV	&1B	VDU extension
TickerV	&1C	Centisecond clock ticks
UpcallV	&1D	OS_UpCall
ChangeEnvironmentV	&1E	OS_ChangeEnvironment

Table 20.2. The OS Vectors.

Writing Code which Intercepts Vectors

Code which intercepts vectors should, ideally, behave as much like the normal vector routine as possible. So, most or all registers should be preserved, the stack should be left uncorrupted, the processor mode left unchanged and so forth. Where SWIs are involved, it is important to preserve the contents of R14 on the stack as it will otherwise be corrupted.

Vector handling code returns, either by intercepting the call or by passing it on to the previous claimant. To intercept the call, return using:

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

so as to retrieve the address placed there by the OS. To pass on the call, copy the contents of R14 on entry back into the PC using:

```
MOV PC, R14
```

(assuming R14 is still valid – if you have preserved R14 elsewhere then clearly a different mechanism will be needed.)

SWIs Which Deal with Vectors

The most fundamental SWIs relating to vectors are OS_Claim and OS_Release – they allow you to attach or detach your routine from the front of the list of claimants for a given vector.

OS_Claim (SWI &1F)

On entry, the vector number to be intercepted should be placed in R0 (taken from the list above), the address of your routine in R1 and a pointer to the private workspace for the routine in R2. The private workspace pointer allows R12 to be initialised where the claimant is a module.

```
3380  MOV R0,#&14      ;Buffer insert vector
3390  ADR R1,myinsv    ;Use address of new routine
3400  SWI "OS_Claim"   ;and claim vector
3410  MOV R0,#&15      ;Buffer remove vector
3420  ADR R1,myremv
3430  SWI "OS_Claim"
3440  MOV R0,#&16      ;Buffer count/purge vector
3450  ADR R1,mycnpv
3460  SWI "OS_Claim"
```

No results are returned by this call.

OS_Release (SWI &20)

This SWI takes the same parameters as OS_Claim and removes the specified claimant from the list associated with the specified vector – preventing subsequent calls to it. The printer buffer similarly releases the vectors.

```
3570  MOV R0,#&14      ;Buffer insert vector
3580  ADR R1,myinsv    ;Extra entry
3590  SWI "OS_Release";Release
3600  MOV R0,#&15      ;Buffer remove vector
3610  ADR R1,myremv
3620  SWI "OS_Release"
3630  MOV R0,#&16      ;Buffer count/purge vector
3640  ADR R1,mycnpv
3650  SWI "OS_Release"
```

OS_CallAVector (SWI &34)

This SWI calls the vector whose number is in R9 on entry, the parameters specific to the vector being provided in R0 to R8. The SWI is useful for calling routines which have no entry point other than through the vector, eg, InsV. Listing 20.2 is an example of the use of this SWI.

```
 10  REM >List20/2
 20  REM by Nicholas van Someren
 30  REM for the Dabhand Guide to the Archimedes OS
 40  REM (c) Copyright AvS and NvS 1988
 50  :
 60  DIM code% 100
 70  P%=code%
 80  [
 90  .start
100  MOV R9,#&16          ;Count and purge vector
110  MOV R1,#3            ;Printer buffer
120  MOV R3,#0            ;Count rather than purge
130  MOVS R4,R1,LSL #1    ;Ensure carry is clear
140  SWI "OS_CallAVector"
150  MOV R0,R1            ;Put the count in R0
160  MOV PC,R14
170  ]
180  REM Send something to the printer.
190  :
200  VDU 2
210  PRINT "HELLO"
220  VDU 3
230  :
240  REM Read the count, clear the buffer, then read the
250  REM count again.
260  :
270  PRINT "Before clearing the buffer:"USR code%
280  *FX 15
290  PRINT "After clearing the buffer: "USR code%
```

Listing 20.2. Using OS_CallAVector.

In some cases it will be useful for a vector to be intercepted *after* the default Operating System routine has been called. This is not directly possible, but can be achieved by using some tricky coding. Acorn recommend the following code fragment for this purpose:

```
.intercept_code
STMFD R13!,{R9}
ADR R9,continue_after + mode
STMFD R13!,{R9,R12}
MOV PC,R14
LDMFD R13!,{R9,R12}
.continue_after
< Code to execute after default routine has been
    called should be placed here >
LDMFD R13!,{PC}
```

The code should be linked to the vector by intercepting it in the normal way. The routine places a dummy return address on the stack and then calls the

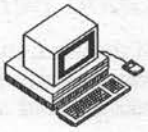
default vector routine. This routine would normally return directly to the main program which called the vector in the first place. However, because we have modified the stack, it actually returns back into our code. We can then do any processing required before returning directly back to the vector caller ourselves.

Remember that the default routine will probably return results in the registers or the flags. These are available to our routine to modify before passing them back to the caller. Care must be taken to preserve any results required. This is particularly easy to overlook if the results are returned in the status flags. Listing 20.3. is a trivial example of intercepting a vector. It intercepts the ReadC vector which is called each time a character is read from the input stream. Before reading the character it prints a "*" as a 'prompt'. The result is that a '*' is output before each character read in.

```
 10  REM >List20/3
 20  REM Example of Intercepting a vector
 30  REM The ReadC vector
 40  REM Archimedes OS: A Dabhand Guide
 50  REM (c) Mike Ginns 1988
 60  REM
 70  :
 80  :
 90  DIM ReadC_intercept 1024
100  P%= ReadC_intercept
110  [
120  :
130  STMFD R13!,{R14}
140  SWI 256+42
150  LDMFD R13!,{R14}
160  MOVS PC,R14
170  ]
180  :
190  SYS "OS_Claim",4,ReadC_intercept,0
```

Listing 20.3. Intercepting ReadC.

21 : Interrupts and Events



Interrupts allow the flow of instruction processing to be diverted (usually temporarily) as a result of an external signal. The ARM CPU caters for two levels of interrupt – normal Interrupt ReQuests (IRQs) and Fast Interrupt reQuests (FIQs), each of which has a set of 'shadow' registers to allow low interrupt latency. Each kind of interrupt may be selectively enabled or disabled by setting the flag associated with that particular interrupt type in the status part of the ARM PC.

When an interrupt is received, the CPU finishes the execution of the current instruction and saves the PC in the link register R14 associated with the particular interrupt. It then disables the relevant interrupt, by setting the relevant flags, and calls the interrupt service routine by jumping through the appropriate vector.

The two classes of interrupt are treated slightly differently: since FIQs are designed for very fast operations, their vector appears last in the hardware vector table so that a branch is not necessary. The FIQ code must therefore reside in the space between the hardware vector at &1C and address &FC so as not to corrupt the OS workspace beyond &FC.

The interrupt service routine needs to determine which device has caused the interrupt and then deal with it as necessary. Clearly this is entirely device dependent. Finally, the interrupt service routine needs to return to the point of interruption, which is achieved by the following instruction:

```
SUBS PC, R14, #4
```

(The offset of four is applied because the value in R14 is one instruction ahead of the actual instruction to execute).

Usually, you will not need to create interrupt service routines of your own because the OS provides a well-defined system for passing unknown interrupts around to applications using the vector claim call OS_Claim, which was discussed earlier.

Good Behaviour

In general, interrupt routines should keep interrupts disabled (the default state) while being executed. This is to prevent the service routine from being called again before it has finished dealing with the current interrupt.

Most SWIs may safely be called from within interrupt routines, provided that the contents of R14_SVC are saved before the call. Acorn recommends the following code to do this:

```
MOV R9, PC          ;save the current PC & mode in R9
ORR R8, R9, #3      ;use R9 to make R8 a supervisor version
TEQP R8, #0         ;use R8 to change mode
MOVNV R0, R0        ;no-op ** REQUIRED **
STMFD R13!, {R14}   ;stack the supervisor R14
SWI swi_number      ;call the SWI
LDMFD R13!, {R14}   ;get R14 back
TEQP R9, #0         ;and restore the original state
MOVNV R0, R0        ;no-op ** REQUIRED **
```

The no-op instructions are needed because changing processor mode can change register banks and the result of access to banked registers is not defined for one cycle.

Obviously, it is silly for FIQ service routines to call SWIs as this pretty much defeats the purpose of using FIQs.

Switching Interrupts On and Off

It is possible to globally enable and disable interrupts by means of the SWIs OS_IntOn and OS_IntOff which neither take nor return any parameters. Clearly the use of these SWIs should be avoided, as they have marked (and usually undesirable) effects on the operation of the OS. An example of the use of these SWIs is given below in listing 21.1.

```
 10 REM >List21/1
 20 REM by Nicholas van Someren
 30 REM Archimedes OS: A Dabhand Guide
 40 REM (c) Copyright AvS and NvS 1988
 50 :
 60 :
 70 REM Turn interrupts off and loop for a while.
 80 :
 90 SYS "OS_IntOff"
100 PRINT"Testing..."
110 T%=TIME
120 FOR x=1 TO 100000
130 NEXT
140 :
```

```
150  REM Turn interrupts on again.
160  :
170  SYS "OS_IntOn"
180  :
190  PRINT"That seemed to take ";(TIME-T%)/100;" seconds."
```

Listing 21.1. Turning interrupts on and off.

The Main Interrupt Vector

It is very unlikely that you will need to intercept the IRQ interrupt system at a lower level than that provided by the normal Operating System vectors. However, if it is vital that your routine is called before *any* other when an interrupt occurs, then the primary interrupt vector can be intercepted.

This vector is a sort of intermediary between the hardware interrupt vectors and the Operating System. It is located in memory at address &100. When an IRQ interrupt occurs, the Operating System jumps to a routine, the address of which is stored at location &100.

Normally, the default routine jumped to from this vector is the Operating System's first level interrupt handler (FLIH). This is responsible for determining the source of the interrupt and calling an appropriate handling routine in the OS.

If users place the address of their own code at location &100, then this will be executed, instead of the default routine, whenever an IRQ interrupt occurs.

It is obviously a very drastic action to try to replace the entire Operating System interrupt handling routine. For this reason, most intercept code will perform its function and then jump on to the default routine. It will, therefore, have to remember the default address which was stored at location &100 when it was initialised.

Events

Events are essentially 'sanitised' versions of interrupts. They are used to inform the user whenever the Operating System performs or detects some significant task or occurence. For example, a key is pressed, a buffer becomes empty or a timer expires. The user may specify that he/she is interested in a particular event. The Operating System will then inform when the event occurs by jumping, via the event vector, to a section of code supplied by the user.

Many events are produced in response to the Operating System detecting an interrupt from the hardware. They allow the user to take action when these significant happenings occur, without having to intercept the interrupt system and deal with the hardware itself.

Twelve events are currently defined for the OS and are detailed in table 21.1. You will notice that most of these events are similar to those used by the BBC MOS.

Event Number	Cause of the Event	Event entry information (R0 = Event number)
0	An output Buffer has become empty	R1 = Buffer number
1	Input buffer was already full	R1 = Buffer number R2 = Character which couldn't be inserted
2	A Character has been placed in an input buffer	R2 = ASCII value of new character
4	Vsync : scanning beam has reached bottom of screen	
5	Interval timer crossed zero	
6	Escape condition detected	
7	RS423 receiving error	R1 = Serial device status R2 = Character received
8	Event generated by Econet	
9	Event generated by the user	
10	Mouse button has changed state	R1 = Mouse X co-ordinate R2 = Mouse Y co-ordinate R3 = Mouse button state R4 = Lower 4 bytes of the real time centisecond value
11	Key Pressed/Released Event	R1 = 0 if key pressed R1 = 1 if key released R2 = Key matrix number
12	A sound event has occurred	R1 = Sound level (always 2) R2 = 0

Table 21.1. The 12 OS Events.

In order to make use of an event, we must first intercept the event vector. The event vector is number &10 and may be intercepted, like any other, using SWI "OS_Claim".

After the vector has been intercepted the attached routine will be called whenever an enabled event occurs. To allow it to differentiate between events, the routine is entered with the event identifying number in R0. Other parameters are passed in various registers as described above.

To enable an event, so that when it occurs the event vector is called, we use the following *FX command.

*FX 14,<n>

Where 'n' is the number of the event which we want to enable. Similarly to disable an event again, we use:

*FX 13,<n>

Obviously, in an assembler program, the SWI OS_Byte version of these two *FX's would be used. For example, to enable event number four we could use the following code:

```
MOV R0,#14    \ Using OSBYTE 14
MOV R1,#4     \ Event number 4
SWI OS_Byte   \ Call OSBYTE
```

On exit from either OS_Byte, R1 indicates the previous state of the event in the following way:

R1 = 0 Event was previously disabled
R1 > 0 Event was previously enabled

As stated previously, most events are triggered by the Operating System detecting an interrupt. The same rules concerning the writing of interrupt handling routines also apply to writing event handlers.

To return after an event routine is completed, we simply use:

```
MOVS PC,R14
```

This is possible as the OS will have set up a return address in R14 for use. It will then deal with the more complex problem of returning to the interrupted code. Note that, like interrupt routines, event handlers *must* preserve R14_SVC before using a SWI instruction. The code to do this was given earlier in the interrupt section.

The user can generate his/her own event by using:

```
SWI "OS_GenerateEvent"
```

On entry, R0 should contain the number of the event which you require to generate. Event number 9 has been specifically reserved as a 'User Event'. R1,R2,R3 etc can contain event parameters to pass to the event handling

routine through the event vector. Listing 21.2 is an example of an event driven routine. It sets up the event vector and then enables event number four. This is the screen vertical sync event (VSync) and means that the routine is entered 50 times a second synchronously with the fly back period of the screen display.

All that the routine does, when called, is to increment the co-ordinates of the mouse pointer. If the pointer reaches the edge of the screen then the increment is reversed and the pointer moves off in the opposite direction.

The result of this is that the mouse pointer moves and bounces around the screen under interrupt/event control. The user is free to enter a command or do what he/she likes as if the process was not running.

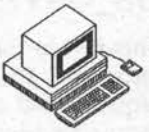
```
 10 REM >List21/2
 20 REM Example of using events
 30 REM Bouncing mouse pointer
 40 REM Archimedes OS: A Dabhand Guide
 50 REM (c) Mike Ginns 1988
100 :
110 DIM Vsync_event 1024
120 FOR pass = 0 TO 3 STEP 3
130 P%= Vsync_event
140 :
150 [
160 OPT pass
170 :
180 CMP R0,#4                \ Check Vsync event
190 MOVNES PC,R14             \ Exit if not vsync
200 :
210 STMFD R13!,{R0-R12,R14} \ Preserve registers on stack
220 :
230 MOV R9,PC                 \ Store PC flags in R9
240 ORR R8,R9,#3              \ Copy flags to R8 selecting SVC mode
250 TEQP R8,#0                \ Write modified flags back into PC
260 MOVNV R0,R0               \ NOP instruction to sync reg banks
270 STMFD R13!,{R14}          \ Preserve R14_SVC on stack
280 :
290 :
300 LDR R0,xpos               \ Modify x position of pointer
310 LDR R1,xinc
320 ADD R2,R0,R1
330 CMP R2,#1216
340 RSBHI R1,R1,#0
350 STR R2,xpos
360 STR R1,xinc
370 :
380 LDR R0,ypos               \ Modify y position of pointer
390 LDR R1,yinc
400 ADD R3,R0,R1
410 CMP R3,#960
```

```
420 RSBHI R1,R1,#0
430 STR R3,ypos
440 STR R1,yinc
450 ADD R3,R3,#64
460 :
470 ORR R2,R2,R3,LSL#16
480 :
490 MOV R0,#3                \ Save pointer x,y in parameter block
500 STRB R0,OSWORD_block
510 STR R2,OSWORD_block+1
520 :
530 ADR R1,OSWORD_block      \ Invoke OSWORD to move pointer
540 MOV R0,#21
550 SWI "OS_Word"
560 :
570 LDMFD R13!,{R14}         \ Restore R14_SVC from stack
580 TEQP R9,#0               \ Restore mode and flags from R9
590 MOVNV R0,R0              \ NOP instruction to sync reg banks
600 :
610 LDMFD R13!,{R0-R12,R14} \ Restore entry registers from stack
620 MOVS PC,R14              \ Return to system
630 :
640 .xpos                    \ X position of pointer
650 EQUD 10
660 .ypos                    \ Y position of pointer
670 EQUD 10
680 .xinc                    \ amount of movement in X direction
690 EQUD 8
700 .yinc                    \ amount of movement in Y direction
710 EQUD 8
720 :
730 EQUW 0                   \ Skip 3 bytes
740 EQUB 0
750 .OSWORD_block            \ 1 non word aligned byte followed by
760 EQUD 0                   \ a single word-aligned word.
770 EQUD 0
780 :
790 ]
800 NEXT
810 :
820 *POINTER
830 SYS "OS_Claim" ,16,Vsync_event,1 : * Link to event vector
840 : * Enable Vsync event
850 *FX14 4
860 :
870 VDU19,0,24,180,150,255
880 PRINT ''"Back in command mode"'
890 PRINT '"Pointer is moving under interrupt control"
```

Listing 21.2. The bouncing mouse pointer.

Notice how the code presented earlier to preserve R14_SVC is actually used in the program. Try using *FX13,4 and *FX14,4 to disable and enable the event which drives the routine. You should see the pointer stop and start appropriately.

22 : Conversion SWIs



One aspect of the OS, which is likely to be appreciated by every programmer, is its range of useful conversion SWIs for dealing with the various formats for strings, numbers, dates and so forth. This chapter summarises these SWIs concisely – you will probably find yourself using them a great deal!

String Conversion and Decoding

These SWIs perform conversions on strings for the OS to use. In particular, they expand ESCAPE characters entered using the split-bar '|' syntax for control characters and the angle-bracket '<>' syntax for OS variables and embedded integers. You will notice that each SWI builds upon the features of the preceding ones.

OS_GSInit (SWI &25)

This call must be made once before using either of the following two string conversion SWIs. On entry, R0 should point to the string to be converted and R2 should have its top three bits set as follows:

Bit	Effect if Set
31	Inverted commas around strings are not stripped out
30	Split-bar control codes are not translated
29	The first space is treated as the terminator

R0 and R2 are returned with values appropriate for calling the other SWIs.

OS_GSRead (SWI &26)

This call gets the next character from a string initialised by OS_GSInit. It is used by the next SWI, OS_GSTrans. On entry it takes the values of R0 and R2 set up by OS_GSInit and returns the desired character in R1, updating R0 and R2 as necessary. OS_GSRead performs the actual translation of the various elements of syntax mentioned earlier – decoding, for example, control characters denoted by split-bar sequences.

OS_GSTrans (SWI &27)

This SWI is the general-purpose string translation routine – it calls GS_Init once and then GS_Read for each character in the string. On entry, it requires R0 to point to the string to be translated, which should be terminated by any of zero, line feed (10) or carriage return (13). R1 should point to a destination buffer with R2 containing its maximum size (and the top three bits set as for OS_GSInit). The resultant string is placed in the buffer, with R0 pointing to the terminator and R2 containing its length. The Overflow flag 'V' is set if the string could not be translated and the Carry flag 'C' is set if the buffer overflowed.

Listing 22.1: A string is set up at string% containing ordinary characters, a character defined as a binary number and the control code for a 'bell'. This is printed out character-by-character using OS_GSRead, and then all in one go using OS_GSTrans to decode the string and OS_WriteN to print it.

```
  10  REM >List22/1
  20  REM GSDemo
  30  REM Archimedes OS: A Dabhand Guide
  40  REM (c) Copyright AvS and NvS 1988
  50  :
  60  REM Reserve some space for string and make
  70  REM $string% the string to be translated.
  80  :
  90  DIM string% 100,work% 100
 100  $string%="Character 254 looks like ""<2_11111110>"" and a
bell sounds like this. <7>"
 110  PRINT $string%
 120  :
 130  REM Prepare to read translated bytes and keep
 140  REM reading characters until carry is set.
 150  :
 160  SYS "OS_GSInit",string% TO where%,,flags%
 170  REPEAT
 180  SYS "OS_GSRead",where%,,flags% TO
where%,nextchar%,flags%;checklast%
 190  IF (checklast% AND 2)=0 VDU nextchar%
 200  UNTIL checklast% AND 2
 210  PRINT
 220  :
 230  REM Translate and print the string in one go.
 240  :
 250  SYS "OS_GSTrans",string%,work%,100 TO ,,L%
 260  SYS "OS_WriteN",work%,L%
 270  PRINT
 280  END
```

Listing 22.1. Using GS calls.

ASCII to Binary Conversions

OS_ReadUnsigned (SWI &21)

This call converts an unsigned string in a given number base to a 32-bit binary number. On entry, R0 should contain the number base (in the range two to thirty-six), R1 should point to the string and R2 should contain the maximum permissible value.

The string may contain valid digits and characters, which may be preceded with '&' for hexadecimal or with 'base_' for a given base. The string is analysed up to the first invalid character for the given base. So, for example, the string '89A' will return 89 if decimal is specified.

The top three bits of R0 contain flags which allow various range checks:

Bit	Effect if set
31	Terminator must be less than ASCII 33
30	Value must be in range 0 to 255
29	Value must be in range 0 to R2

The call returns with R1 pointing to the end of the string (if it was valid), or is unaltered (if the string was invalid). R2 contains the converted value, or zero if an error arose. The Overflow flag 'V' will be set if the string was not in the specified format.

OS_EvaluateExpression (SWI &2D)

This very sophisticated SWI translates a string expression which may include a whole range of arithmetic, logical and string operations. On entry, R0 points to the source string, R1 points to a buffer for the translated result and R2 contains the maximum size of that buffer. On return, the type of the result is indicated by R1, which is zero for an integer result and non-zero for a string (whose actual length is returned in R2, with R0 intact). If the size of the buffer is exceeded the error 'Buffer overflow' is generated.

OS_EvaluateExpression applies OS_GSTrans to the string in order to evaluate any parameters in angle-brackets, and treats any sequences of letters which are not operators as variable names. Valid numeric and logical operators are:

```
+   -   *   /   MOD
=   <>  >=  <=  <   >
>>      >>>     <<
AND OR EOR NOT
```

Strings enclosed in inverted commas may be used, and the following string operators are valid:

Operator	Effect/value
+	Concatenation of two strings
RIGHT n	As BASIC RIGHT$
LEFT n	As BASIC LEFT$
LEN	Length of string
STR n	Translate number to string
VAL "x"	Translate string to number

Binary to ASCII Conversions

OS_BinaryToDecimal (SWI &28)

This SWI converts a signed 32-bit integer to an ASCII decimal string which is placed in a buffer. On entry, R0 contains the integer, R1 points to the buffer and R2 contains its maximum size. On return, R1 is preserved (so it points to the converted string) and R2 contains the length of the string. The 'Buffer overflow' error is generated if the resultant string will not fit into the buffer supplied. Listing 22.2 at the end of this section demonstrates the use of OS_BinaryToDecimal. OS_Write0 is used to display the string returned.

```
 10 REM >List22/2
 20 REM BintoDec
 30 REM by Nicholas van Someren
 40 REM Archimedes OS: A Dabhand Guide
 50 REM (c) Copyright AvS and NvS 1988
 60 REM Set up a string buffer.
 70 :
 80 DIM buffer 100
 90 :
100 REM Convert -123456 into a signed string and display it.
110 :
120 SYS "OS_BinaryToDecimal",-123456,buffer,100 TO ,,length
130 buffer?length=0
140 SYS "OS_Write0",buffer
150 PRINT
160 END
```

Listing 22.2. Binary to Decimal Conversion.

OS_Convert Group (SWIs &D0-&EA)

This range of SWIs is responsible for conversions between integers and ASCII strings of various types. Each SWI takes a value in R0, a pointer to a buffer in R1 and the maximum buffer size in R2.

There are several forms of each SWI, to deal with input and output values of different sizes – in each case the appropriate digit is appended to the SWI name given below.

On return, R0 now points to the buffer, R1 to the terminating character of the string within the buffer and R2 contains the number of bytes left in the buffer. The 'Buffer overflow' error is generated if the resultant string will not fit into the buffer supplied.

OS_ConvertHex1/2/4/6/8 (SWIs &D0-&D4)

These SWIs produce a hexadecimal result of the specified number of digits, with leading zeros added as necessary. Note that an ampersand '&' is not added to the start of the string.

OS_ConvertCardinal1/2/3/4 (SWIs &D5-&D8)

These SWIs produce an unsigned decimal result using the specified number of bytes of the input value – no leading zeros are added to the start of the string.

OS_ConvertInteger1/2/3/4 (SWIs &D9-&DC)

These SWIs produce a signed decimal result (ie, with a leading minus sign '–' if relevant) using the specified number of bytes of the input value – no leading zeros are added.

OS_ConvertBinary1/2/3/4 (SWIs &DD-&E0)

These SWIs produce an ASCII binary string result using the specified number of bytes of the input value with leading zeros added.

OS_ConvertSpacedCardinal1/2/3/4 (SWIs &E1-&E4)

These SWIs are the same as the ConvertCardinal group except that they insert a space between every three digits of the result, eg, 811679 becomes 811 679.

OS_ConvertSpacedInteger1/2/3/4 (SWIs &E5-&E8)

These SWIs are the same as the ConvertInteger group except that they also insert a space between every three digits.

SWI Name and Number Conversions

Two SWIs are provided which convert a SWI name to its number and vice versa. The standard SWI naming structure is used, viz:

{X}ChunkName_Name

The exact case of each letter of a SWI name is as important as ever.

OS_SWINumberToString (SWI &38)

On entry, R0 contains the SWI number to be converted, R1 points to a buffer and R2 contains the maximum size of the buffer. The call looks up the textual form of the SWI and, if possible, returns it as a null-terminated string in the buffer pointed to by R1.

OS_SWINumberFromString (SWI &39)

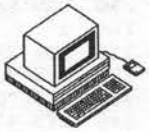
On entry, R0 points to a string terminated by an ASCII value of less than 33. The call looks up the SWI number for the given string and returns it in R0. The 'X' error bit 17 and the SWI chunk number will have been added to the SWI number automatically, so this call deals with all relevant cases.

```
10  REM >List22/3
20  REM NumberConv
30  REM by Nicholas van Someren
40  REM Archimedes OS: A Dabhand Guide
50  REM (c) Copyright AvS and NvS 1988
60  DIM R% 100
70  :
```

```
 80  REM For each of the converstion SWIs, find the name of
 90  REM the conversion and use it to convert &8904EA12.
100  :
110  FOR I%=&D0 TO &E8
120  SYS "OS_SWINumberToString",I%,R%,100
130  SYS "OS_Write0",R%
140  SYS &120
150  SYS I%,&8904EA12,R%,100
160  SYS "OS_Write0",R%
170  SYS "OS_NewLine"
180  NEXT
190  END
```

Listing 22.3. Demonstrating number conversion.

23 : Miscellaneous SWIs



This chapter discusses the use of a number of SWIs which do not fall neatly into any of the groups described in previous chapters, but which nevertheless, serve a useful purpose. They are listed here in groups according to function and are presented in alphabetical order within the groups. Each SWI description is accompanied by an example program to illustrate its function more clearly.

Timer Functions

Several SWIs exist that allow routines to be called at regular intervals or after a specified delay. These routines expect to be given a time specification (as appropriate), a pointer to a routine to execute and a value to place in R12 when executing the routine (to simulate the workspace pointer for modules). These SWIs are:

SWI "OS_CallAfter"	SWI &3B
SWI "OS_CallEvery"	SWI &3C
SWI "OS_RemoveTickerEvent"	SWI &3D

SWI "OS_CallAfter" (SWI &3B) Execute Routine after Time Delay

This call takes a delay in centiseconds in R0, the address of a routine to be executed in R1 and the value to be passed in R12 to the routine in R2. A timer is set up by the OS and the code will be executed after the specified delay. This call returns (almost) immediately with no return parameters.

SWI "OS_CallEvery" (SWI &3C) Repeatedly Execute Routine

This call is similar to the above, but it calls the routine repeatedly at the interval specified in centiseconds in R0, the other two parameters being the same as those for OS_CallAfter. Once initiated, this call to the routine will occur at the specified interval until it is terminated in software (see below)

or the machine is reset. The routine should be as short as possible and, in general, behave as if it were an interrupt service routine.

SWI "OS_RemoveTickerEvent" (SWI &3D) Shut Down Timed Execution

This call takes the address of a routine in R0, and in R1 the value which is passed to the routine's R12 (the two consistent parameters from the above calls). It uses this information to identify and remove a timed event from the internal lists of the OS.

In listing 23.1 you can see each of these SWIs being used. The example creates a series of counters which are incremented at different speeds according to the interval time selected for each.

SWI "OS_ReadMonotonicTime" (SWI &42) Read Time Since Power-on

This call returns the number of centiseconds which have elapsed since the computer was switched on. It takes no entry parameters and returns the time in R0. Listing 23.2 demonstrates this.

```
 10 REM >List 23/1
 20 REM CallAftEv
 30 REM by Nicholas van Someren
 40 REM Archimedes OS: A Dabhand Guide
 50 REM (c) Copyright AvS and NvS 1988
 60 DIM code% 1000,counters% 16
 70 P%=code%
 80 [
 90 .countcode ;A simple routine
100 STMFD R13!,{R0}     ;Preserve R0 on stack
110 LDR R0,[R12]        ;Load what is pointed to by R12
120 ADD R0,R0,#1        ;Increment it
130 STR R0,[R12]        ;Store it back
140 LDMFD R13!,{R0}     ;Restore R0
150 MOV PC,R14 ;Return
160 ]
170 :
180 REM Zero all the counters
190 :
200 !counters%=0
210 counters%!4=0
220 counters%!8=0
230 counters%!12=0
240 :
250 REM Use OS_CallEvery to call the counter code
260 REM at three different rates.
```

```
270 :
280 SYS "OS_CallEvery",5,countcode,counters%
290 SYS "OS_CallEvery",10,countcode,counters%+4
300 SYS "OS_CallEvery",17,countcode,counters%+8
310 :
320 REM Keep showing the counters until Space is pressed.
330 :
340 PRINT''"Press Space Bar to stop..."''
350 WHILE INKEY(-99)=0
360 VDU 13
370 PRINT !counters%,counters%!4,counters%!8;
380 ENDWHILE
390 :
400 REM Turn the counters off again.
410 :
420 SYS "OS_RemoveTickerEvent",countcode,counters%
430 SYS "OS_RemoveTickerEvent",countcode,counters%+4
440 SYS "OS_RemoveTickerEvent",countcode,counters%+8
450 :
460 REM Now call the count code after 4 seconds.
470 :
480 PRINT''"Setting up a 'CallAfter' for 4 seconds"''
490 SYS "OS_CallAfter",400,countcode,counters%+12
500 :
510 REM Wait until counter is incremented and show
520 REM the elapsed time while we are waiting.
530 :
540 TIME=0
550 WHILE counters%!12=0
560 VDU 13
570 PRINT;TIME/100;SPC5;
580 ENDWHILE
590 T%=TIME
600 PRINT''"Counter incremented."
610 END
```

Listing 23.1. Using time function SWIs.

```
 10 REM >List23.2
 20 REM ReadMonoTm
 30 REM Archimedes OS: A Dabhand Guide
 40 REM (c) Copyright AvS and NvS 1988
 50 :
 60 SYS "OS_ReadMonotonicTime" TO A%
 70 PRINT"This machine has been on for ";A%DIV360000;" hours
";(A%DIV6000)MOD 60;" minutes and ";(A%DIV100)MOD 60;" second(s)."
 80 END
```

Listing 23.2. Reading time elapsed since power-on.

VDU Related SWIs

While we have consciously chosen not to repeat the documentation of the VDU drivers in this book, a number of SWIs have been provided which perform useful extraneous functions. They are rather a mixed bag, so further explanation is left to the descriptions of the individual calls.

SWI "OS_CheckModeValid" (SWI &3F) Check Whether a Given Mode is Available

This call tests to see if a given screen mode, supplied in R0, can be accommodated by the memory available. If it can, the Carry flag 'C' is cleared and R0 is preserved; otherwise the Carry flag is set and R0 contains either –1 if no such mode exists or –2 if there is insufficient memory to support it. Also, R1 is returned containing the mode number which the VDU would use instead (its best approximation).

Listing 23.3 issues this SWI for each of the modes from 0 to 23 and displays the results returned.

```
 10  REM >List23/3
 20  REM by Nicholas van Someren
 30  REM Archimedes OS: A Dabhand Guide
 40  REM (c) Copyright AvS and NvS 1988
 50  :
 60  REM Go through all the modes, checking whether
 70  REM they can be selected.
 80  :
 90  FOR mode%=0 TO 23
100  SYS "OS_CheckModeValid",mode% TO status%,usemode%
110  PRINT"Mode ";mode%;
120  CASE status% OF
130  WHEN -1:PRINT" is not available; would use mode ";usemode%
140  WHEN -2:PRINT" is too big; would use mode ";usemode%
150  OTHERWISE:PRINT" is OK"
160  ENDCASE
170  NEXT
180  END
```

Listing 23.3. Checking screen mode.

SWI "OS_RemoveCursors" (SWI &36)
Remove the Cursor(s) from the Display
SWI "OS_RestoreCursors" (SWI &37)
Restore the Cursor(s) to the Display

These SWIs turn the display cursor on and off. They neither take nor return any parameters. This is an alternative form of the VDU 23,0,0/1 process. Listing 23.4 demonstrates this in action.

```
10  REM >List23/4
20  REM by Nicholas van Someren
30  REM Archimedes OS: A Dabhand Guide
40  REM (c) Copyright AvS and NvS 1988
50  :
60  MODE 0
61  REPEAT
70  SYS "OS_RemoveCursors"
80  PRINT"Cursor off - press a key"
90  key=GET
100 SYS "OS_RestoreCursors"
110 PRINT"Cursor on - press a key"
111 key=GET
120 UNTIL FALSE
```

Listing 23.4. Cursor related SWIs.

SWI "OS_Mouse" (SWI &1C)
Return Mouse Information

This call returns the oldest entry in the OS mouse buffers. It takes no parameters, but returns the mouse X and Y co-ordinates in R0 and R1, the state of the mouse buttons in R2 and the absolute time at which the reading was taken in R3. The buttons are flagged in the bottom three bits of R2, with bit 0 being the right-hand button, bit 1 the middle button and bit 2 the left-hand button. The absolute time is taken directly from the SWI OS_ReadMonotonicTime which is documented above.

Listing 23.5 demonstrates this call by accepting a series of movements and button depressions and plotting lines interconnecting the locations where the buttons were pressed.

```
10  REM >List23/5
20  REM by Nicholas van Someren
30  REM Archimedes OS: A Dabhand Guide
40  REM (c) Copyright AvS and NvS 1988
50  :
60  MODE 0
```

```
 70 :
 80 REM Turn the pointer on, get a few entries into
 90 REM the mouse buffer and wait for Return.
100 :
110 *POINTER
120 PRINT"Move the pointer to different places on the screen"
130 PRINT"and press a mouse button on each. Press Return"
140 PRINT"when you have finished."
150 REPEAT UNTIL GET=13
160 :
170 REM Display up to 20 positions.
180 :
190 CLS
200 FOR n=0 TO 20
210 SYS "OS_Mouse" TO x,y,z,t
220 DRAW x,y
230 PRINTTAB(0,0);z;",";t;SPC10
240 wait=INKEY(100)
250 NEXT
260 *POINTER 0
270 END
```

Listing 23.5. Reading the mouse buffer.

SWI "OS_ReadModeV" (SWI &35) Read Mode Variables

This call allows a number of internal VDU variables to be read for a given mode (which is not actually selected). On entry, R0 should contain the chosen mode and R1 the number of the variable (from the list below). When the call returns, R2 will contain the value of the specified variable or the Carry flag 'C' will be set if either of the parameters were invalid.

The VDU variables which may be extracted are listed below in table 23.1.

Number	Meaning
0	Mode indicators
	Bit 0 clear = graphics mode
	set = text only mode
	Bit 1 set = teletext mode
	clear = non-teletext mode
	Bit 2 set = interline gap in this mode
	clear = no gap in this mode
1	Number of text columns in this mode minus one
2	Number of text rows in this mode minus one
3	Maximum logical colour (1, 3, 15 or 63)

Number	Meaning
4	Horizontal pixel resolution 0=1280 1=640 2=320 3=160
5	Vertical pixel resolution 1=512 2=256
6	Number of bytes per row of pixels
7	Number of bytes used by this mode
8	Shift factor for row start address To find the offset to the start of row Y from the start of screen memory add (Y<<Shift factor)*5
9	Log to base two of number of bits per pixel
10	Log to base two of number of bytes per pixel

Table 23.1. The VDU variables.

Example 23.6 demonstrates the use of this SWI by extracting the number of columns, rows, colours and the size of each mode from 0 to 21 and displays the results as a table.

```
 10 REM >List23/6
 20 REM ReadModeV
 30 REM by Nicholas van Someren
 40 REM Archimedes OS: A Dabhand Guide
 50 REM (c) Copyright AvS and NvS 1988
 60 MODE 0
 70 :
 80 REM Display information about screen modes.
 90 :
100 PRINT"Mode","Cols.","Rows","Colours","Size"'
110 FOR mode%=0 TO 21
120 SYS "OS_ReadModeVariable",mode%,1 TO ,,cols
130 SYS "OS_ReadModeVariable",mode%,2 TO ,,rows
140 SYS "OS_ReadModeVariable",mode%,3 TO ,,colours
150 SYS "OS_ReadModeVariable",mode%,7 TO ,,size
160 PRINT ;mode%,;cols+1,;rows+1,;colours+1,;size/1024;"K"
170 NEXT
180 END
```

Listing 23.6. Read VDU variables for given mode.

SWI "OS_ReadPalette" (SWI &2F)
Read Palette Values

This call reads the palette settings for a particular logical colour (supplied in R0) for a specific part of the display (supplied in R1). The value in R1 should be 16 for a normal colour, 24 for a border colour or 25 for a cursor colour.

The results, in the form of the first flashing colour and the second flashing colour, are returned in R2 and R3. If the colour is steady then these numbers will be the same. Each is a four byte value, with the bottom byte containing control information and the remaining three indicating the amount of red, green and blue respectively. The control byte contains one of the following values:

Value	Meaning
0-15	BBC compatible colour
16	Steady colour specified as RGB
17-18	Mark or Space flashing colour specified as RGB

Listing 23.7 demonstrates the use of this SWI by presenting the palette definitions for each logical colour in a specified display mode.

```
10 REM >List23/7
20 REM by Nicholas van Someren
30 REM Archimedes OS: A Dabhand Guide
40 REM (c) Copyright AvS and NvS 1988
50 :
60 REM Set the print field to 8 and input screen mode.
70 :
80 @%=8
90 INPUT"Which screen mode?"mode%
100 MODE mode%
110 PRINT"Colour","Red","Green","Blue","Sup'cy"
120 FOR col%=0 TO 15
130 SYS "OS_ReadPalette",col%,16 TO ,,mark%,space%
140 :
150 REM If mark%=space% then colour is steady,
160 REM otherwise, colour is flashing.
170 :
180 IF mark%=space% THEN
190 red$=STR$((mark%>>8)AND&FF)
200 green$=STR$((mark%>>16)AND&FF)
210 blue$=STR$((mark%>>24)AND&FF)
220 IF mark% AND &80 sup$="Y" ELSE sup$="N"
230 ELSE
240 red$=STR$((mark%>>8)AND&FF)+","+STR$((space%>>8)AND&FF)
250 green$=STR$((mark%>>16)AND&FF)+","+STR$((space%>>16)AND&FF)
260 blue$=STR$((mark%>>24)AND&FF)+","+STR$((space%>>24)AND&FF)
```

```
270 IF mark% AND &80 sup$="Y" ELSE sup$="N"
280 IF space% AND &80 sup$+=",Y" ELSE sup$+=",N"
290 ENDIF
300 PRINT STR$col%,red$,green$,blue$,sup$
310 NEXT
320 END
```

Listing 23.7. Using OS_ReadPalette.

SWI "OS_ReadPoint" (SWI &32) Read Colour of Pixel

This call allows the colour and tint of a pixel to be read. The co-ordinates of the pixel must be supplied as (R0,R1) with the results returned in R2-R4 as follows:

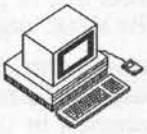
Register	Information
R2	Colour of pixel
R3	Tint of pixel
R4	Validity flag (0=valid, –1=off the display)

Listing 23.8 demonstrates this SWI by printing some colour bars and allowing the pointer to be moved about the display, printing the colour and tint found at each pixel.

```
 10 REM >List23/8
 20 REM by Nicholas van Someren
 30 REM Archimedes OS: A Dabhand Guide
 40 REM (c) Copyright AvS and NvS 1988
 50 :
 60 REM Set up screen and trap errors tidily.
 70 :
 80 MODE 8
 90 OFF
100 ON ERROR ON:OSCLI"POINTER 0":END
110 FOR col%=129 TO 132
120 COLOUR col%
130 PRINT STRING$(80," ")
140 NEXT
150 REM Turn the pointer on, and display the colour of
160 REM the pixel on which the pointer is located.
170 :
180 *POINTER
190 REPEAT
200 MOUSE x,y,z
210 SYS "OS_ReadPoint",x,y TO ,,colour,tint,flag
220 PRINTTAB(30,14)colour,tint;
230 UNTIL FALSE
```

Listing 23.8. Reading a pixel.

24 : The ARM Chip Set



In order to produce an overall cost-effective computer system Acorn needed more than just a cheap processor, they needed the supporting circuitry to go with it. Shortly after the design of the ARM was finished Acorn's VLSI design team designed three chips to complete a workable computer. These chips became known as MEMC, VIDC and IOC and are a memory controller, a video controller and an I/O controller respectively.

MEMC is responsible for refresh and address multiplexing for the low-cost high-density dynamic RAM which Acorn chose to design in. It also deals with memory protection in different processor operating modes. Finally, it stores the addresses of the memory used for the video display, cursor generation and the sound output, providing Direct Memory Access (DMA) control for those features.

VIDC is responsible for turning three streams of data, fed it by MEMC, into a video picture and stereo sound, containing the colour palette and providing all the timing of video synchronisation signals.

IOC is a versatile input/output controller, comprising a number of counter/timers, several bi-directional input/output lines and both edge and level-sensitive interrupt lines.

The rest of this section is devoted to an examination of some of the internal details of these chips. Because one of the purposes of an Operating System is to render it unnecessary to program such devices directly, we have chosen not to examine their register structure in too much detail. If you really must change their contents yourself, rendering your programs dangerous to most users, you can get hold of the relevant data books directly from VLSI Technology, the foundry which mass produces them.

Inside MEMC

MEMC is a device which may only be addressed; data to be written to MEMC is encoded onto the address lines to reduce the pin count. This is done by mapping MEMC into a very large section of the memory map and using the lower order address bits to carry the data information, while the higher

order bits of the address carry the information of what to do with the data. The MEMC registers can only be written to from ARM Supervisor Mode.

MEMC controls the memory map of the Archimedes, ensuring that certain parts of the memory map, including the part it occupies itself, can only be accessed in the appropriate processor mode. Furthermore, it divides the processor address space into a "Logical", or imaginary area, and a "Physical" area, some of which may actually be real memory (RAM chips). By translating logical addresses into physical ones, through a series of tables, MEMC is able to keep the address space consistent through changes of the amount of memory and programmed display sizes in use.

This memory mapping system provides the basis for a genuine "virtual" memory system, but current versions of the Operating System are not able to take advantage of this fully. Other Operating Systems, notably UNIX, are well designed to suit this sort of addressing scheme, although the MEMC implementation has some non-trivial problems.

The memory map for MEMC is illustrated in figure 24.1.

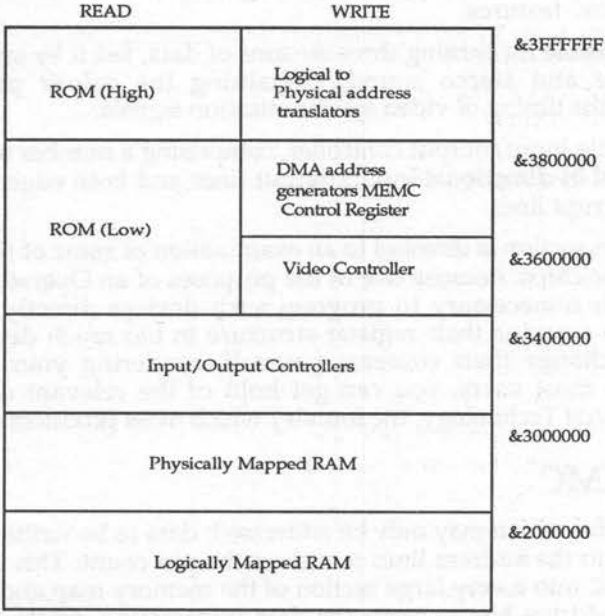


Figure 24.1. The MEMC memory map.

The memory in both the logically and physically mapped areas is divided into a number of "pages". MEMC provides for up to 128 pages of memory and these pages can be 4, 8, 16 or 32k in size, all being the same chosen size. The memory in the physically mapped area corresponds directly to the dynamic RAM chips, for which MEMC provides the addressing and refresh timing.

Virtual Memory Support

Each page in the logical memory area can be programmed, either to correspond to one of the 128 pages of physical RAM or is considered to be "paged out", in which case it has no corresponding physical page. In the latter case, if the processor tries to access any word in the inaccessible page, an "Address Exception" is caused. This allows software to be added to provide a virtual memory system, ie, the user program (and the higher levels of the Operating System) believe that there is a full complement of 32Mbs of RAM in the system and can happily access any address in the logical area. If the logical address has no corresponding physical address, then the address exception would be trapped by software and the lower level Operating System would find the appropriate page of memory, usually somewhere on a hard disc. Thus, just a few megabytes of RAM and a 32Mb hard disk can appear to be 32Mbs of RAM, albeit rather slower than the real thing.

The MEMC Control Register

Most features of MEMC are controlled by the MEMC control register. This register is programmed by making up an address according to the data to be programmed, and then writing (anything, the data is ignored) to that address. The address layout is shown in figure 24.1 below:

```
Address Bits (25-0)

25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1  1  0  1  1  x  1  1  1  x  x  x  t  o  s  v d d h h l l p p x x
```

Figure 24.2. The MEMC Control Register

Key to MEMC Control Register

- x Don't care
- t Test mode, this bit must be zero
- o Operating System mode, used to protect RAM from user programs:

0 Program User Mode
1 Operating System User Mode

s Sound DMA bit
0 Disabled
1 Enabled

v Video DMA bit
0 Disabled
1 Enabled

d Dynamic RAM refresh bits
00 No Refresh
01 Refresh only during video flyback
10 No Refresh
11 Continuous refresh

h High ROM access time
00 450 ns
01 325 ns
10 200 ns
11 200 ns with 60ns nibble-mode

l Low ROM access time
00 450 ns
01 325 ns
10 200 ns
11 200 ns with 60ns nibble-mode

p Page Size
00 4k
01 8k
10 16k
11 32k

The Logical to Physical Address Translator

As we saw above, memory accesses to addresses below 32Mbs (&2000000) are translated through the logical to physical address translator. This part of MEMC is very closely coupled to the Operating System and should never be programmed by user programs. If you are curious about its workings, the details of its operation can be obtained from the data sheet on MEMC which is available from VLSI Technology.

DMA Address Generators

Part of MEMC's function is to provide Direct Memory Access (DMA) addresses for VIDC. VIDC is not connected to the address bus and so can not

output addresses for data it requires. Instead, these addresses are generated by MEMC which then freezes the processor while it directs RAM to put the data onto the bus, allowing VIDC to read it. The DMA address registers are programmed with the physical addresses of the information and so should only be modified by the Operating System. The important VIDC registers are:

Video	Vinit, Vstart, Vend and an invisible Vptr
Cursor	Cinit
Sound	Sstart, SendN and Sptr

and their formats are as follows:

```
25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1  1  0  1  1  x  n  n  n  Physical Address   Physical Address x x
```

Bits 16-2 refer to bits 18-4 of the physical address to be used; this address must lie on a 16 byte boundary, in order that MEMC can retrieve the data efficiently from RAM which is using a fast "nibble mode".

The start of screen address (the logical address of the memory location of the top left-hand corner of the screen) can be programmed "legally" by using OS_Word &22.

Inside VIDC

VIDC is the exact opposite of MEMC in that it has no connections to the address bus at all, only connections to the data bus. Register selection is achieved using the higher order bits of the 32-bit data word, while the actual data is in the lower order bits. VIDC triggers the transfer of information at addresses generated by MEMC using several control lines which interconnect the chips. When any of the buffers in VIDC become nearly empty, a MEMC controlled data transfer is requested and the buffers reloaded.

The major programmable aspect of VIDC, as far as user programs are concerned, is the colour palette, which translates logical colours stored in memory into physical colours which are actually displayed. There are three, four-bit Digital to Analogue Convertors (DACs) which drive the output connector, allowing 4096 different colours to be displayed. In 2, 4 and 16 colour modes these DACs may be used to generate the relevant number of colours by loading them with values in the format shown below:

Palette registers (as seen in 2, 4 or 16 colour modes):

S b3 b2 b1 b0 g3 g2 g1 g0 r3 r2 r1 r0

S	Supremacy bit, allowing video mixing with external hardware
b3, b2, b1, b0	Blue colour control bits
g3, g2, g1, g0	Green colour control bits
r3, r2, r1, r0	Red colour control bits

In the 256 colour modes the top four bits of the logical colour are sent directly to the colour DACs as shown below. The bottom four bits index into sixteen palette registers which provide the remaining eight bits of data needed by the DACs, also shown below:

Palette registers (as seen in 256 colour modes):

S d7 b2 b1 b0 d6 d5 g1 g0 d4 r2 r1 r0

S	Supremacy bit, allowing video mixing with external hardware
b2, b1, b0	Blue colour control bits
g1, g0	Green colour control bits
r2, r1, r0	Red colour control bits
d7, d6, d5, d4	Direct data bits from logical colour

By default, the 256 colour palette is arranged so that :

d3=b2
d2=r2
d1=r1,g1 & b1
d0=r0,g0 & b0

ie, the 8-bit data becomes %BGgRbrTt, where Rr, Gg, Bb and Tt are two bit numbers representing levels of Red, Green, Blue and Tint (whiteness).

Sound Frequency and Stereo Position

VIDC will reproduce up to eight channels of sound through two logarithmic DACs which provide a stereo effect. The data for each channel is supplied in successive bytes, whose transfer is triggered by the sound buffer becoming nearly empty. Each channel has a stereo position register associated with it and the stereo effect is created by pulse width modulation of the analogue level between the left and right channel DACs.

Inside IOC

IOC is a versatile interface and peripheral control chip which contains a number of programmable interrupt masks, input/output lines and four counter/timers. The OS sets up these timers to perform various internal functions:

- Timer 3 generates the baud rate for the serial keyboard interface.
- Timer 2 is used to generate baud rates for the RS423 serial controller, which needs an external generator to achieve disparate transmit and receive rates.
- Timer 0 is used by the Operating System to generate the $^1/_{100}$ second interrupts which are used to time the system clock, colour flash rate and keyboard repeat rate (amongst others!).

The remaining timer (timer 1) can be used for any purpose and is free for user programs. All the timers have the same register layout and are programmed similarly. Each timer may be loaded with a 16 bit value which is decremented by one every 500nS. When the counter value reaches zero an interrupt will be generated if the control registers have been set appropriately and the counter re-loaded.

The following piece of assembler should allow you to set up Timer 1 to generate interrupts which may be caught via the IrqV vector (see the chapter on Vectors). On entry, R0 should contain the counter initialisation value in the lower 16 bits.

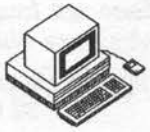
```
.UseCounter
;R0 contains the value for the counter in low 16 bits
LDR R2,IOC_Base
STRB R0,[R2,#&50]    ;Store low byte
MOV R0,R0,LSR #8
STRB R0,[R2,#&54]    ;Store high byte
MOV R0,PC            ;Get the User Mode PSR
SWI "OS_EnterOS"
MOV R1,PC            ;Get the supervisor mode PSR
ORR R1,R1,#3<<26     ;Mask out interrupts
TSTP R1,R1           ;Set the PSR flags
LDRB R3,[R2,#&12]
ORR R3,#%01000000    ;or use BIC to clear bit
STRB R3,[R2,#&10]
TSTP R0,R0           ;Restore the PSR flags
STR R0,[R2,#&58]     ;Start the counter
...
.IOC_Base
 EQUD &03400010
```

And here is a listing of a sample interrupt handler which checks to see if it is indeed the timer which has caused an interrupt:

```
.InterruptCode
STMFD R13!,{R0,R2}
LDR R2,IOC_Base
LDRB R0,[R2,#&16]
ANDS R0,R0,#%01000000
BNE OurInterrupt
LDMFD R13!,{R0,R2}
MOV PC,R14                ;Don't claim

.OurInterrupt
...
your code here
...
LDMFD R13!,{R0,R2,PC}^    ;Claim interrupt
```

25 : Floating Point Model



Besides the integer arithmetic and data processing instructions we have seen, the ARM was designed with a general-purpose 'co-processor' interface which allows its instruction set to be expanded. This mechanism was provided primarily to allow hardware support for floating point operations, which are more commonly performed by complex software packages.

The ARM Co-processor Interface provides a two-level mechanism for expanding the instruction set. There exists an 'undefined instruction' vector, through which ARM will jump if it encounters an instruction which is unrecognised by the ARM itself and is not supported by any additional hardware present. The Co-processor Interface allows hardware to be added to deal with new instructions, but where the hardware is absent, the undefined instruction vector provides the means for a software emulator to take its place (albeit more slowly). This is precisely the mechanism employed for floating point operations.

The Operating System contains a module called the 'Floating Point Emulator' (the FPE) which deals with floating point arithmetic. If a floating point co-processor is fitted, the FPE is pre-empted and the operations are actually completed by hardware. Whether hardware is present or not, however, the software interface (that is, the programmer's view) remains the same, thus providing an important 'transparency' and ensuring that the same software can be used in either situation.

Unfortunately, one hurdle remains: the ARM BASIC Assembler does not support floating point instructions (which were still being defined when the assembler was written) and thus they must either be hand-assembled or called through a more sophisticated assembler (yet to be announced).

Floating Point Programmer's Model

The ARM floating point system conforms to the IEEE specification which is in common use throughout the computer industry. The ARM programmer's model provides eight floating point registers, known as F0 to F7, and a number of precision formats which allow the programmer to trade off

speed against accuracy. Whilst the format used internally is not defined, there are four formats for storage of floating point numbers in memory which are outlined below:

IEEE Single precision (S)
32 bits: 1 sign bit
23-bit mantissa (fraction)
8-bit exponent

IEEE Double precision (D)
64 bits: 1 sign bit
52-bit mantissa (fraction)
11-bit exponent

Double extended precision (E)
96 bits: 1 sign bit
64-bit mantissa (fraction)
15-bit exponent
16 bits not used

Packed decimal BCD (P)
96 bits: 1 sign digit
19-digit mantissa
4-digit exponent

The ARM floating point model also provides for a floating point status register to indicate the status of calculations. This register includes the following flags:

Overflow
Underflow
Division by zero
Inexact result
Invalid operation

Flags indicating the result of floating point operations are automatically copied into the ARM status register by means of the Co-processor Interface, thus keeping the programmer's job as simple as possible.

The ARM Floating Point Instructions

The ARM co-processor interface defines three distinct types of instruction, all of which are used in the floating point system. They are:

CPDT	Co-processor data transfer (for moving values to and from memory)
CPRT	Co-processor register transfer (for moving between ARM and FP registers)
CPDO	Co-processor data operation (for initiating co-processor operations)

The ARM floating point system supports a number of actual instructions for each of these instruction types. The rest of this section details all of these instructions.

Co-processor Data Transfer

Syntax:

<mnemonic>{cond}<precision> Fx, address

There are two CPDT instructions for floating point data movement:

Mnemonic	Effect
LDF	Load floating point register
STF	Store floating point register

These instructions move data between main memory and the floating point registers. They require the following parameters to be specified:

<precision>	One of S, D, F or P (as discussed earlier)
Fx	One of the floating point registers F0-F7
address	Either [Rn]{,#offset} or [Rn,#offset]{!}

The offset is from the ARM base register specified and is in the range –1020 to +1020. The offset is added to the base register when write-back is specified with pre-indexed addressing and is always added when post-indexed addressing is used.

Co-processor Register Transfer

Syntax:

varies - see below

There are six CPRT instructions for floating point register movement. Two are concerned with conversions between floating point and integer values; the remaining four deal with the floating point control and status registers. The CPRT instructions are summarised overleaf:

Mnemonic	Effect	Operation Performed
FLT	Change integer to floating point	Fx:=Rd (or Fx:=#value)
FIX	Floating point to integer	Rd:=Fx
WPS	Write FP status	FPSR:=Rd
RFS	Read FP status	Rd:=FPSR
WFC	Write FP control	FPC:=Rd*
RFC	Read FP control	Rd:=FPC*

* Supervisor Mode only

The status and control operations simply require an optional condition and a register name. The conversion instructions are a little more complex – their full syntax is shown below:

FLT{cond}<precision>{rounding mode} Fx,(Rd #value)
FIX{cond}<precision>{rounding mode} Rd,Fx

The precision field is of the same form as discussed earlier. The rounding mode controls how the value is rounded for conversion, according to the table below:

Mode	Letter	
Nearest	–	(default rounding mode)
Plus infinity	P	(always round up)
Minus infinity	M	(always round down)
Zero	Z	(round towards zero)

Co-processor Data Operations

Syntax:

<unaryop>{cond}<precision>{rounding mode} Fd,(Fx #value)
<binaryop>{cond}<precision>{rounding mode} Fd,Fx,(Fy #value)

The co-processor data operations are divided into two groups: unary operations (which take one parameter) and binary operations (which take two). In either case, one parameter may be a floating point constant from the list below:

Floating point constants: 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 0.5, 10.0

The following tables summarise the unary and binary floating point operations available.

Unary Floating Point Operations

Mnemonic	Effect	Calculation performed
MVF	Move	Fd := Fx
MNF	Move negated	Fd := –Fx
ABS	Absolute value	Fd := ABS (Fx)
RND	Round to integer	Fd := integer value of Fx
SQT	Square root	Fd := square root of Fx
LOG	Log to base ten	Fd := log base ten of Fx
LGN	Log to base e	Fd := log base e of Fx
EXP	Exponent	Fd := e^Fx
SIN	Sine	Fd : = sine of Fx
COS	Cosine	Fd := cosine of Fx
TAN	Tangent	Fd := tangent of Fx
ASN	Arc sine	Fd := arc sine of Fx
ACS	Arc cosine	Fd := arc cosine of Fx
ATN	Arc tangent	Fd := arc tangent of Fx

Binary Floating Point Operations

Mnemonic	Effect	Calculation Performed
ADF	Add	Fd := Fx+Fy
MUF	Multiply	Fd := Fx*Fy
SUB	Subtract	Fd := Fx-Fy
RSF	Reverse subtract	Fd := Fy-Fx
DVF	Divide	Fd := Fx/Fy
RDF	Reverse divide	Fd := Fy/Fx
POW	Power	Fd := Fx^Fy
RPW	Reverse power	Fd := Fy^Fx
RMF	Remainder	Fd := Fx MOD Fy
FML	Fast multiply	Fd := Fx*Fy[1]
FDV	Fast divide	Fd := Fx/Fy[1]
FRD	Fast reverse divide	Fd := Fy/Fx[1]
POL	Polar angle	Fd := polar angle of Fx, Fy

[1] Single precision result

Note that the 'fast' operations produce a result whose accuracy is that of a single-precision calculation, regardless of the specified precision of the instruction.

Rounding of the trigonometric functions only takes place after the calculation has been completed, thus preserving as much accuracy as possible during the calculation. The 'nearest' rounding mode is used.

Co-processor Status Transfer Instructions

Syntax:

<mnemonic>{cond}<precision>{rounding mode} Fx, Fy

The co-processor status transfer instructions allow comparisons to be made between values in floating point registers and, according to the result of such comparisons, the setting of the ARM's own status flags. The co-processor status transfer instructions are:

Mnemonic	Effect	Calculation performed
CMF	Compare FP	Compare Fx with Fy
CNF	Compare negated FP	Compare Fx with –Fy
CMFE	Compare FP (exception)	Compare Fx with Fy
CNFE	Compare negated FP (exception)	Compare Fx with –Fy

The comparisons are available in two forms: normal and exception. The exception form raises an exception (error) if either of the two parameters are invalid numbers (that is, invalid by IEEE standards). Such values could be the result of the failure of previous calculations, for example. The IEEE standard specifies that CMF should be used for equality comparisons (that is, where a BEQ or BNE will be used after the instruction) and CMFE should be used for all others (that is, where a 'greater than' or 'less than' test will follow).

The ARM PSR flags are set according to the result of these floating point operations and indicate the following states:

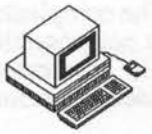
Flag	Meaning
N	Less than – Fx was less than Fy (or –Fy)
Z	Equal
C	GT or equal – Fx was greater than or equal to Fy (or –Fy)
V	Unordered

Conclusion

The ARM Floating Point Co-processor instruction set provides a wealth of operations for high-precision arithmetic. When a hardware Floating Point Co-processor is not present, a software emulator performs the operations 'transparently' from the programmer's point of view. The ARM BASIC V assembler does not support the floating point instructions which must, therefore, be hand-coded or assembled with another assembler.

The complexity of the IEEE floating point arithmetic standard is such that it is not possible to give full details in a general-purpose book such as this. The interested reader is referred to Acorn Floating Point Co-processor documentation and to the IEEE standard definition document.

A : Programs Disc



A Programs Disc to accompany this book is available direct from Dabs Press. It contains all the example programs listed in these pages plus several extra utility programs. The disc is supplied with its own manual for ease of use.

The bonus programs include:

ListSWIs	A program to list all the SWI names in a particular SWI number chunk.
Find	A utility program which will locate a file or group of files that are 'lost' somewhere on a disc. The file specification, which may include wildcards, and directory path may be specified.
DirCopy	A utility program which copies whole directories from one disc to another on a single-drive machine. It has the file type of an application because it uses all the RAM in a machine to speed up the copying of directory structures and their files. The transfer is typically achieved with only two disc swaps on an Archimedes 310.
DescRMA	A development utility which is useful when developing programs which use the RMA (eg, modules). It simply provides a printout of the RMA description returned by the SWI called OS_Module.

The Floating Point Assembler

FPA allows floating-point instructions to be included in machine-code programs assembled using the BASIC V assembler. BASIC V does not usually allow floating-point mnemonics to be assembled, so the FPA adds extensions in the form of a BASIC LIBRARY which permit this.

The new pseudo mnemonics that are added are:

FEQ<S>const	Assembles a 4-byte single precision constant.
FEQD const	An 8-byte double precision one.

FEQE const A 12-byte high-precision one.

FEQP const A 12-byte packed BCD one.

Examples of using these are also included on the disc.

The disc is available in 3.5in ADFS format and the programs are not copy protected in any way, so you are free to integrate them into your own software as it develops. The disc is compatible with all versions of the Archimedes and both Arthur and RISC OS Operating Systems.

To obtain your copy of the Archimedes Operating System programs disc send £9.95 to the address given below. Cheques and POs should be made payable to Dabs Press. Access and Visa card orders are acceptable by phone, simply by quoting your card number, type and expiry date – and don't forget your address!

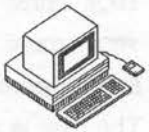
By post:

Dabs Press,
76 Gardner Road,
Prestwich,
Manchester,
M25 7HU

By phone:

061-773 2413

B : Dabhand Guides Guide



Books and Software for the Archimedes

Dabs Press already have a list of books, software and games for the Archimedes and this is being expanded to include a wide range of Archimedes products. Those already in an advanced stage of preparation are detailed in the following pages. Please note that all details are correct at the time of writing but are subject to change without notice. Please phone or write to confirm availability before ordering.

All future publications are in an advanced state of preparation. Content lists serve as a guide, but we reserve the right to alter and adapt them without notification. Publication dates and contents are subject to change. All quoted prices are inclusive of VAT (on software; books are zero-rated), and postage and packing (abroad add £2 or £10 airmail). All are available from your local dealer or bookshop or, in case of difficulty, direct from Dabs Press. If you would like more information about Dabs Press books and software, then drop us a line at 5 Victoria Lane, Whitefield, Manchester, M25 6AL, and we'll send our latest catalogue. See previous page for ordering details.

Archimedes Books

Archimedes Assembly Language

By Mike Ginns. Price £14.95. Spiral bound 368 pages.
ISBN 1-870336-20-8. Available now. **NEW** Second Edition
Programs disc £9.95 – £21.95 inclusive when ordered with book.

This popular and informative book is now in its Second Edition and includes coverage of RISC OS.

This is a complete guide to programming the Archimedes in machine code. Mike Ginns provides a clear, step-by-step account of using the assembler using simple, but useful, programs and provide the practice to illustrate the theory, thus making it ideal for the beginner. But this guide goes much further. For instance, it explains how to use the Debugger and there is a

large section on implementing BASIC equivalents in machine code, plus coverage of Arthur and RISC OS and using SWIs, WIMPs and fonts.

The powerful Operating System is covered with details of how to access its many facilities from the machine code level. Within this are details of using graphics, sound, windows, the font painter and the mouse, all from within machine code programs. To make the transition from BASIC to machine code as painless as possible, the book contains a large section on implementing BASIC statements in machine code.

A programs disc accompanies the book which contains all of the various programs used in the text, plus 11 extra utility programs including a disassembler, various memory manipulators and a disc sector editor. The programs disc is supplied with its own 16 page manual.

Reviews:

Risc User, July/August 1988: *"The style of the text throughout the book is easy to read...I would recommend Archimedes Assembly Language."*

Archive, August 1988: *"The actual explanations are lucid...Overall then, this is a comprehensive and wide ranging book which stands up well as both a tutorial on assembly language and as a guide to the programming environment and facilities provided by Arthur. I recommend it..."*

C : A Dabhand Guide

By Mark Burgess. Price £14.95. Perfect bound 512 pages.
ISBN 1-870336-16-X. Available now. **NEW** Second Edition
Programs disc £9.95 – £21.95 inclusive when ordered with book.

PCW Said: *"...I only wish it had been available when I was learning C."*

A behind-the-scenes storm has quietly been sweeping over the microcomputer world during the last few years: it is the C programming revolution. So much so that all the popular micros now have C compilers available to them.

Spread over an amazing 512 pages, this thoroughly readable Dabhand Guide leaves you in no doubt as to the natural language in which to program your computer. From elementary principles, PCW contributor and author, Mark Burgess introduces the C philosophy in a highly readable, no nonsense manner. Step by step, page by page you ascend the C ladder with simple illustrated and documented programs.

But why should you want to learn C at all? The answers are many, not least compatibility, portability and speed. C is a general purpose language.

It can be used to write any kind of program from an accounting package to an arcade game. It has sophisticated facilities built in which are quite unlike those of any other language. The range of C commands span from a higher level than BASIC to as low a level as machine code. C holds nothing back from the programmer – there are virtually no limitations.

C is a standard language – programs the world over are written to this standard and in such a way as to allow them to be transferred to other machines and run again, in many cases with little or no editing required. A source program written in C on the Amstrad PC, for instance, would generally compile and run quite happily on the Archimedes, the Amiga, or any other PC for that matter.

Speed – a vital factor in the running of programs – is assured because a C program is compiled into ultra fast machine code. Write your very own commands in a friendly environment and let the C compiler transform it into machine code – no assembly language need be known! And what's more the original C source program remains intact for re-editing or fine tuning as you require.

Thirty-seven chapters, six appendices, a glossary and a comprehensive index make **C: A Dabhand Guide** probably *the* guide to programming in C. Included is a chapter on programming in C on the Archimedes, (and BBC and the Master 128/Master Compact for that matter).

Unique diagrams and illustrations help the reader to visualise programs and to think in C. Assuming only a rudimentary knowledge of computing in a language such as BASIC or PASCAL, the reader is provided with a grounding in how to build up programs in a clear and efficient way.

To help the beginner a complete chapter on fault finding and debugging assists in tracing and correcting both simple and complex errors.

A Programs Disc is available for most of the major micros, and this contains the listings in the book plus several other useful utilities including an adventure game and an indexer. The extra programs are documented in an informative manual.

The first review of **C: A Dabhand Guide** appeared in Beebug Magazine in June 1988 and it had this to say: "*The 512 pages cover all important aspects of C...the tone is friendly and the explanations are full and easy to understand without being patronising...the program structure diagrams which illustrate the larger programs are very helpful...the book being full of good advice about program design and layout. In conclusion, then, a very good, reasonably priced introduction to C for the non-specialist.*"

BASIC V

By Mike Williams. Price £9.95. Perfect bound 140 pages approx.
ISBN 1-870336-75-5. Available February 1989.

This book provides a practical guide to programming in BASIC V on the Acorn Archimedes and covers BASIC V on RISC OS. Assuming a familiarity with the BBC BASIC language in general, it describes the many new commands offered by BASIC V, already acclaimed as one of the best and most structured versions of the language on any micro, and is illustrated with a wealth of easy-to-follow examples throughout.

An essential aid for all Archimedes users, it will also appeal to existing BASIC users who wish to be conversant with its many new features. BASIC V includes several new control structures which are major innovations. These are discussed and the text is littered with simple but effective examples. For the graphics programmer, the new extended graphics commands are covered with interesting examples of their use along with control and manipulation of the colour palette.

Other major topics covered include:

- WHILE, IF and CASE
- Use of mouse and pointer
- Local error handling.
- Sound
- The Assembler
- Matrix operations
- Functions and Procedures
- Operators and string handling
- Arthur and RISC OS
- Programming hints and tips

The author, Mike Williams, is Editor of Beebug magazine and Risc User, the largest circulation Archimedes specific magazine.

Archimedes Software

Archimedes Basic Compiler

By Paul Fellows. Price £99.95 Inclusive. Available Now **NEW** Version 2.
Two discs. 148 page Reference Guide, 56 page User Guide.
Demo Disc available for £2. Refundable on full order. Supports over 100 BASIC V commands and supplied with sample programs.

ABC: The fast and powerful way to write instant machine code!

If you want it all – speed, power and performance, then look no further than the Archimedes Basic Compiler. ABC takes programs written in BASIC V and transforms them into superfast ARM machine code. Speed increases

of 5000% are possible depending on the nature of the program being compiled.

A&B Computing said: *"ABC is a vital part of the programmer's toolbox, it puts compilers on other systems to shame. Unquestionably one of the most impressive pieces of software I have yet seen running on the Archimedes."*

Archive Magazine said: *"I can tell you now, I am very impressed. This is a superb package, which I thoroughly recommend..."*

Main Features:

- Converts BASIC V programs to ARM machine code
- Completely stand-alone code—does not access BASIC ROM
- All compiled code is position independent
- Speed increases of over 5000% possible
- Code size ratio approx. 1:1 (against tokenised source)
- Target program size limited only by disc space
- Conditional compilation supported
- Supports floating point arithmetic (using FPE)
- CALL and USR parameter passing greatly enhanced
- New pseudo-variables included
- Runs on any Archimedes
- Friendly window-based control system
- Relocatable module making option
- Application, utility and service module types supported
- Full in-line assembler support
- Compiles using disc or RAM or both
- Execute immediately after compilation
- Large range of compiler directives
- Manifest constants implemented for extra speed
- Comprehensive and interesting examples disc
- Intelligent disassembler produces source of compiled code
- No additional runtime modules required
- No intermediate code system
- 148pp Reference Guide and 56pp User Guide
- ARM fp processor compatible
- RISC OS and Arthur 1.2 compatible
- Technical support to registered users
- Absolutely no royalties to pay on compiled code

Version 2 now supports the following additions and improvements:

- Double/Extended precision floating point
- RETURN parameters
- Multiple-ENDPROCs and function returns

- LOCAL Errors
- Scope rules
- Extended Compiler Directives

ABC is written by Paul Fellows, head of the Acorn team which wrote the original Archimedes Operating System. Complete specification available on request.

Instigator

By Mike Ginns. Price £49.95. Available March 1989.

The RISC OS Compatible Archimedes System Manager

Instigator is a powerful extension to your Archimedes Operating System – Arthur 1.2 and RISC OS – and the ideal foil for programmers and software developers alike. Containing over 65 * commands this module provides a wide range of exciting and invaluable system aids.

Instigator provides an extremely powerful working environment for the user. It allows tasks be carried out quickly and efficiently. You get on with the task in hand, Instigator provides the necessary information for you and works with the Operating System to carry out your wishes, whether you are using application packages or programming the machine itself.

Its new commands and facilities will prove indispensable to any serious user of the Archimedes system. The commands make new operations possible, help to simplify the use of existing features and give the user unprecedented control over the machine.

Instigator Commands

Here is a list of some of the commands provided by Instigator:

*Medit, *Mmove, *Mfill, *Mfind, *Diss, *Tidy, *Compare, *Blist, *Dimmer, *Half, *Full, *VIDC, *OpenWindow, *CloseWindow, *Files, *Confirm, *FSsave, *FSload, *Compress, *Uncompress, *Printer, *CSD, *SetPath, *Paths, *UsePath, *KillPaths, *SavePaths, *LoadPaths, *Return, *RGB, *Colours, *SetPalette, *Palette, *ListPalette, *KillPalette, *SavePalette, *LoadPalette, *Keys, *SaveKeys, *LoadKeys, *SaveOSVars, *LoadOSVars, *OSVars, *SaveCMOS, *LoadCMOS, *IStatus, *Dedit, *Dget, *Dput, *Dsearch, *FreeMap, *LineEdit, *Archive, *History, *Cut, *Definemode, *Hourglass, *Percentage, *Xinfo, *Smooth

Instigator is supplied on disc as a Relocatable Module. An examples disc is included along with a 100 page User Guide.

And There's More!

Dabs Press will be adding to their increasing range of quality books and software for a wide range of micros during the next year. For the **Archimedes** this includes ABC65 a 6502 cross compiler allowing 6502-based code and Sideways RAM images to be generated from BASIC V developed and tested on the Archimedes. The stand-alone 6502 code can then be transferred to a BBC B, B+, Master 128 or Master Compact computer. ABC65 will be available in the second quarter of 1989.

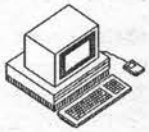
Forthcoming for the **Amiga** is ACE, a BBC BASIC V and Amiga Microsoft BASIC compiler bearing a strong resemblance to ABC but taking particular advantage of the Amiga's own graphics and sound facilities. Books to be released for the Amiga include AmigaDOS: A Dabhand Guide (which covers releases 1.2 and 1.3), Amiga BASIC: A Dabhand Guide and Amiga 500 First Steps.

The **Z88** will be supported with Z88: A Dabhand Guide and Z88 PipeDream: A Dabhand Guide. The former is an indispensable guide for all Z88 users while the later is the Z88 wordprocessors companion.

For the **PC** and **Amstrad** PCs our range of books will be extending to include Ability Plus: A Dabhand Guide, Shareware: A Dabhand Guide and PostScript: A Dabhand Guide. These will complement our titles WordStar 1512/Express: A Dabhand Guide and SuperCalc 3: A Dabhand Guide.

For full details on these and other Dabs Press publications, write or phone now for our free and extensive catalogue. See page 303 for address details.

Index



67
$ 67
%*0 50
%0 49
% 67 72 84
& 67
* command 40 47
*/ 48 70
*ACCESS 75
*APPEND 76
*AUDIO 196 199
*BUILD 76
*CAT 77
*CDIR 77
*CHANNELVOICE 197 200
*CLOSE 77
*CONFIGURE 51 59
*COPY 78
*COUNT 79
*CREATE 80
*DELETE 80
*DIR 81
*DUMP 81
*ECHO 52
*ENUMDIR 82
*ERROR 52
*EVAL 53
*EX 82
*EXEC 83
*FX 53
*GO 53 149
*GOS 54
*HELP 54
*IF 55
*IGNORE 55
*INFO 83
*KEY 56
*LCAT 83
*LEX 84
*LIB 73 84
*LIST 84
*LOAD 70 85
*MODULES 120
*OPT 86
*PRINT 86
*QSOUND 198 205
*REMOVE 87
*RENAME 87
*RMCLEAR 120
*RMKILL 120 121 126
*RMLOAD 121
*RMREINIT 121
*RMRUN 121 125
*RMTIDY 122 126
*RUN 48 70 88
*SAVE 88
*SET 49 56 57
*SETEVAL 58
*SETMACRO 58
*SETTYPE 72 76 89
*SHADOW 58
*SHOW 59
*SHUT 89
*SHUTDOWN 89
*SOUND 197 201
*SPEAKER 196 199
*SPOOL 90
*SPOOLON 90
*STAMP 71 90
*STATUS 59
*STEREO 196 200
*TEMPO 198 205
*TIME 60
*TUNING 197 203
*TV 60
*TYPE 91
*UNPLUG 121 122
*UNSET 60
*UP 91
*VOICES 197 203
*VOLUME 197 204
*WIPE 92

- 67
. 67
6502 16 33
: 48 67
< 48 271
> 48 271

ABC 307
add 26
address calculation 20
address exception 255
address validation 156
address, hardware 156
ADFS 66
adjust button 162
ADR 38
Alias$ 49
Alias$LoadType 85
aliases 49
alignment 21 34
allocation, memory 34
amplitude 202
AND 26
anti-aliasing 186 188 193
application environment 150
application top-down 151
application workspace 152 153
application writing 149
application, temporary 151
Archimedes 304
ArgsV 259
arithmetic instruction 26
ARM 16 17 287
arrow icon 162
arrow 173
Arthur 1.2 bugs 155 156
ascending stack 32
ASCII conversion 273 274
assembler 20 24 32 304
assembler, floating point 302
assembly directive 37
assembly error 36
assembly language 304
assembly, offset 35 36
assembly, two-pass 36
attribute 75

bar, double 271
bar, split 271
bar, vertical 271
barrel shifter 24
base number, SWI chunk 136
BASIC V 20 33 307
BGetV 259
binary conversion 273 274
boot options 86
box, rub-out 190
BPutV 259
braces 48
branch 25 34
branch with link 25
branch, return from 26
buffer size, DMA 208
bugs, Arthur 1.2 155 156
button 161
button, adjust 162
button, menu 162
button, mouse 170
button, select 161
ByteV 258

C flag 22
caching, font 186
calculation, address 20
CALL 38
carat 171
caret 68
Carol 87
Carry flag 22
catalogue 77
ChangeEnvironmentV 259
character input/output 238
character input 238
character line 240
character output 244
check mark 173
chunk, memory 153
chunk, SWI 42
CISC 15
claim heap 155
Cli$Prompt 57
clicking 161
CliV 258
close icon 162
CnpV 259

co-ordinate conversion 190
co-ordinate, Font Painter 190
co-ordinate, OS 190 191
co-ordinate, window 163
co-processor 295
code, condition 22 23
command decoding 133
command line interpreter 40 41 47
commands, OS_CLI 50
comments 48
compare 27
comparison instruction 27
compatability, software 156
condition code 22 23
control, keyboard 241
conversion SWIs 271
conversion, ASCII 273 274
conversion, binary 273 274
conversion, co-ordinate 190
conversion, string 271
conversion 275
ConvertBinary 275
ConvertCardinal 275
ConvertHex 275
ConvertInteger 275
ConvertSpacedCardinal 275
ConvertSpacedInteger 275
copy options 78
Copy$Options 78 79
CPDO 297 298
CPDT 297
CPRT 297
create directory 77
CSD 68 72 81
CSL 68 72 83 84
currently selected directory 68
currently selected library 68
cursor 282

DAC 195 291
data abort 255
date stamp 71 90
death, module 126
decoding code, SWI 137
decoding table 133
decoding table, SWI 137
decoding, command 133
decoding, help 133
decoding, string 271
descending stack 32
describe heap 154
DescRMA 302
device filing system 66 69
dimming 173
DirCopy 302
directive, assembly 37
directory 67
directory tree 116
directory, create 77
directory, currently selected 68
directory, examine 82
directory, parent 68
directory, previously selected 68
disc name 68
disc, program 302
display position 60
display screen 286
divide 28
division 28
DMA 18 287 290
DMA buffer size 208
double bar 271
dragging 162 171
DumpFormat 84

empty stack 32
environment 150
environment, application 150
EOR 26
equates 37
error handling, file 102
error numbering 44
error, assembly 36
error, module 123
error, SWI 43
ErrorV 258
ESCAPE 45 76 239 244
evaluation, expression 273
event, vertical sync 268
event-driven 163
EventV 259
event 265
examine directory 82
exception, address 255
execute address 70
execution 38 39
expression evaluation 273
extend heap 155

file error handling 102
file naming 67
file redirection 48
file renaming 87
file type 70 71 72 85 89 113
File$Path 73 101
FileSwitch 66 74
FileV 259
filing system, device 66 69
filing system, RAM 66
filing system 66 69 93
fill 228
finalisation, module 126
FindV 259
Find 302
FIQ 255 263
flag, Carry 22
flag, C 22
flag, icon 165
flag, Negative 22
flag, N 22
flag, Overflow 22 43
flag, overflow 124
flag, V 22 43 124
flag, Zero 22
flag, Z 22
FLIH 265
floating point 295
floating point assembler 302
floating point emulator 295
floating point instruction 295
font caching 186
Font Manager 186
Font Painter 186
font 186
Font Painter co-ordinate 190
Font painting options 189
Font$Prefix 186 187
FontSize 186
Font_CacheAddr 187
Font_ConverttoOS 190
Font_Converttopoints 191
Font_FindFont 187
Font_LoseFont 187
Font_Paint 189
Font_ReadDfn 191
Font_ReadInfo 191
Font_SetFontColours 188
Font_SetFont 189
Font_SetPalette 188
Font_StringWidth 192
foreign character 67
format, module 124
forward reference 35
FPA 302
FPE 295
free 232
frequency, sound 292
FSControlV 259
full stack 32
function key 243

gate off 231
gate on 227
GBPBV 259
guidelines, software 156

handle 165
handling code offset, SWI 136
hardware address 156
hardware vector 255
header, module 124
heap manager 154
heap, claim 155
heap, describe 154
heap, extend 155
heap, initialise 154
heap, memory 153
heap, release 155
help decoding 133
help string, module 132

icon flag 165
icon, arrow 162
icon, close 162
icon, shuffle 162
icon, stretch 162
icon, toggle 162
IEEE 295
immediate value 24
in-line string 249
independence, position 37
indexing 30
indirect string 250
initialisation, module 125
initialise heap 154
input stream 238

input, character 238
input/output, character 238
input/output, simple 238
install 232
instantiate 232
instruction, arithmetic 26
instruction, comparison 27
instruction, floating point 295
instruction, logical 26
instruction, multiple register 31
instruction, unknown 255
InsV 259
intercepting vector 259
interlace 60
interrupt 20 21 22 40 263
interrupt, software 32 255
IOC 18 287 293
IRQ 255 263
IrqV 258

justification, text 189

kbd: 69
Key$ 56 57
keyboard control 241
keyboard status byte 242
keypress 171

label 25 34 35
library 68 72
limit, time 240
line, character 240
link register 20 25
link, branch with 25
list, register 31
ListSWIs 302
load address 70
load register 29
Load$Type 71
logical instruction 26
logical memory 153 288
Logical Work Area 162
LWA 162 163 165

machine code 304
machine RESET 255
macro 58 62
manager, heap 154
map, memory 288
Marvin 63
MEMC 18 287
memory address translation 290
memory allocation 34
memory chunk 153
memory heap 153
memory management 153
memory map 288
memory page 153
memory reservation 34
memory, logical 153 288
memory, physical 153 288
memory, virtual 153 288 289
menu button 162
menu 172 173
metric 186
mode variables 283
mode, screen 283
module 119 123
module death 126
module error 123
module finalisation 126
module format 124
module header 124
module help string 132
module initialisation 125
module start-up 125
module title string 132
module workspace 123
module, printer buffer 141
module, relocatable 119
module, shell 152 157
mouse 161 282
mouse button 170
MouseV 259
multi-tasking 153 164
multiple register instruction 31
multiplication 27 28
multiply 27 28

N flag 22
Negative flag 22
NFS 66
NOT 26
null reason code 167
null: 69
number, SWI 41
numbering, error 44

offset assembly 35 36
options, boot 86
options, copy 78
options, Font painting 189
OR 26
OS co-ordinate 190 191
OSRDCH 239
OSWORD &00 240 241
OS_Args 107
OS_BGet 105
OS_BinaryToDecimal 274
OS_BPut 107
OS_Byte &02 239
OS_Byte &03 245
OS_Byte &05 247
OS_Byte &81 240
OS_Byte &C7 248
OS_Byte SWIs 242
OS_CallAfter 278
OS_CallAVector 260
OS_CallEvery 278
OS_ChangeEnvironment 151 152
OS_CheckModeValid 281
OS_Claim 260
OS_CLI 47 61
OS_CLI commands 50
OS_Convert SWIs 275
OS_EvaluateExpression 53 273
OS_Exit 151
OS_File 93
OS_Find 100
OS_FSControl 109
OS_GBPB 102
OS_GenerateError 45
OS_GenerateEvent 267
OS_GetEnv 150
OS_GSInit 271
OS_GSRead 271
OS_GSTrans 62 134 272
OS_Heap 154
OS_Module 122 138 151 154
OS_Mouse 282
OS_PrettyPrint 250 251
OS_ReadC 239
OS_ReadEscapeState 45
OS_ReadLine 241
OS_ReadModeV 283
OS_ReadMonotonicTime 279
OS_ReadPalette 285
OS_ReadPoint 286
OS_ReadUnsigned 273
OS_ReadVarVal 61
OS_Release 260
OS_RemoveCursors 282
OS_RemoveTickerEvent 279
OS_RestoreCursors 282
OS_ServiceCall 126
OS_SetVarVal 62
OS_SWINumberFromString 42 276
OS_SWINumberToString 42 276
OS_ValidateAddress 156
OS_WriteC 248
OS_WriteI 249
OS_WriteO 250
OS_WriteS 249
output stream 245
output, character 244
Overflow flag 22 43
overflow flag 124

page, memory 153
palette 285 291
parent directory 68
pathname 68
PC 20
physical memory 153 288
Physical Work Area 162
pitch 202
pixel 186 286
pointing 161
point 186
polling 166
polling loop 166
position independence 37
position, display 60
position, stereo 292
post-indexing 30
pre-fetch abort 255
pre-indexing 30
previously selected directory 68
printer buffer module 141
printer stream 246
printer: 69
PRM 14
program counter 20
program disc 302
Programmers Reference Manual 14

PSD 68
PSR 20
punctuation 67
PWA 162 163 165
R14 20 25
R15 20
RAM filing system 66
rawkbd: 69
rawvdu: 69
ReadCV 258
ReadLineV 259
reason code 127 167
reason code, null 167
reason-code 164
redirection, file 48
redraw, window 167
register list 31
register, load 29
register, shadow 263
register, store 29
release heap 155
relocatable module 119
relocatable module area 119
RemV 259
reservation, memory 34
return 26
return from branch 26
RISC 15
RMA 119
ROM 156
root 67
RS423 stream 246
rub-out box 190
Run$Path 73
Run$Type 71
sample rate 208
screen mode 283
screen save 94
screen, display 286
scroll bar 162
scroll offset 165
search path 72
select button 161
selection 161
service call 126
service routine 263
shadow register 263
shell module 152 157
shifting 24
shuffle icon 162
sideways ROM 119
simple input/output 238
Smalltalk 161
software compatability 156
software guidelines 156
software interrupt 32 255
software vector 258
sound frequency 292
Sound SWIs 207
SoundChannels 196
SoundDefault 201
SoundDMA 195 196
SoundScheduler 197
Sound_AttachNamedVoice 215
Sound_AttachVoice 216
Sound_Configure 207
Sound_Control 218
Sound_ControlPacked 218
Sound_Enable 210
Sound_InstallVoice 215
Sound_LogScale 217
Sound_Pitch 217
Sound_QBeat 223
Sound_QInit 221
Sound_QSchedule 221
Sound_QTempo 223
Sound_ReadControlBlock 219
Sound_RemoveVoice 215
Sound_SoundLog 217
Sound_Speaker 210
Sound_Stereo 212
Sound_Tuning 217
Sound_Volume 216
Sound_WriteControlBlock 219
split bar 271
spool file 247
spool stream 247
stack, ascending 32
stack, descending 32
stack, empty 32
stack, full 32
start-up, module 125
status byte, keyboard 242
stereo position 292
store register 29
stream, input 238

stream, output 245
stream, printer 246
stream, RS423 246
stream, spool 247
stream, VDU 246
stretch icon 162
string conversion 271
string decoding 271
string, in-line 249
string, indirect 250
sub-directory 67
sub-menu 173 174
subtract 26
supervisor mode 17 22 32
supremacy 292
SVCB 226
SWI 32 41
SWI chunk 42
SWI chunk base number 136
SWI decoding code 137
SWI decoding table 137
SWI error 43
SWI handling code offset 136
SWI number 41
SWI translation 137
SWIs, conversion 271
SWIs, OS_Byte 242
SWIs, OS_Convert 275
SWIs, Sound 207
SWIs, VDU 281
SWIs, WIMP 175
Sys$ 57
Sys$DateFormat 60

temporary application 151
test 27
text justification 189
tick mark 173
TickerV 259
time limit 240
timer 278 293
title bar 162 165
title string, module 132
toggle icon 162
top-down, application 151
translation, memory address 290
translation, SWI 137
tree, directory 116

TWIN 149 152
two-pass assembly 36

UKPLOTV 259
UKSWIV 259
UKVDU23V 259
unknown instruction 255
unknown PLOT vector 259
unknown SWI vector 259
unknown VDU23 vector 259
up 68
UpCallV 259
update 232
user interface 161
user mode 17 22
USR 38

V flag 22 43 124
validation, address 156
variables, mode 283
VDU extension vector 259
VDU stream 246
VDU SWIs 281
vdu: 69
VDU 186
VDUX 186
VDUXV 259
vector 255 263
vector, hardware 255
vector, intercepting 259
vector, software 258
vector, unknown PLOT 259
vector, unknown SWI 259
vector, unknown VDU23 259
vector, VDU extension 259
vertical bar 271
vertical sync event 268
VIDC 18 287 291
virtual memory 153 288 289
voice generator code 232
voice generators 195
voice generator 226

WIMP 161
WIMP SWIs 175
Wimp_CloseDown 172
Wimp_CloseWindow 169
Wimp_CreateMenu 172 173
Wimp_DeleteWindow 172
Wimp_GetRectangle 168

Wimp_Initialise 164
Wimp_OpenWindow 166 169
Wimp_Poll 166 172
Wimp_RedrawWindow 167 168
Wimp_UpdateWindow 168
window co-ordinate 163
window control block 164 165
Window Manager 161 163 175
window 161
window redraw 167
Wipe$Options 92
WordV 258
workspace, application 152 153
workspace, module 123
write-back 30
WriteCV 258
writing, application 149

X bit 42 44

Z flag 22
Zero flag 22
\ 67
^ 67
{ 48
| 48 271
} 48
67

Notes

A Dabhand Guide

For Archimedes users who take their computing seriously, this guide to the Operating System gives you a real insight into the micro's inner workings. The book is applicable to any model of Archimedes whether running the Arthur or RISC OS Operating Systems.

The Relocatable Module system is one of the many areas covered — its format is explained, and the information necessary to enable you to write your own modules and applications is provided. This tutorial approach is repeated through the book.

The sound system is explained and the text includes much information never before published.

The discerning user will revel in the wealth of information covering many aspects of Arthur and RISC OS including:

- The ARM Instruction Set
- Writing Relocatable Modules
- Writing Applications
- VIDC, MEMC and IOC
- Sound
- The Voice Generator
- SWIs
- Vectors and Events
- Command Line Interpreter
- The FileSwitch Module
- Floating Point Model

and much more...

Throughout the book, programs are used to provide practical examples to use side by side with the text, which go to make this publication *the* table-side companion for all Archimedes users.

Alex and Nic van Someren have both worked for Acorn Computers. Alex is a former Technical Editor of *Acorn User* magazine and author of numerous computer-related books.

£14.95

ISBN 1-870336-48-8

9 781870 336482