2

PROGRAMMER'S REFERENCE MANUAL

Acorn

Archimedes

PROGRAMMER'S REFERENCE MANUAL

Acorn

Archimedes

Designed and laser-typeset by Human-Computer Interface Limited, Cambridge.

First published 1987
Issue 1 November 1987
Published by Acorn Computers Limited
Part number 0476,006

ISBN 1 85250 053 0

CONTENTS

MODULES

A relocatable module is a piece of software which, when loaded into the machine, may behave as a normal application program or, more usefully, as an extension to the operating system. Hence it can contain a language or filing system, receive service calls, add new * commands etc.

Relocatable modules run in an area of memory designated the Relocatable Module Area (RMA), which is maintained by the system. Modules are not guaranteed to be loaded at any particular address. Therefore, their code must be relocatable.

The OS provides facilities to help modules integrate themselves into the system. For example, the operating system normally responds to the *HELP service call for them, extracting any relevant help text automatically so they do not need to respond to it themselves.

Modules are a key feature of the Archimedes software environment. Their discussion has been left until relatively late in this manual because, for the user, the way in which modules work is not very important. The facilities they provide are integrated into the OS using SWIs and * commands in a way which makes them appear to be part of the system.

However, it is clear that any major piece of software written for the A-series machines should be implemented as a module, so this chapter is included as a guide for the writer of such software.

There are several * commands provided by the OS to handle modules, including one to load a module file from the filing system. These commands are described first. Next the SWIs relating to module functions are described. The most important of these is OS_Module. This provides the machine code interface to the functions provided by the module * commands. It also enabled modules to claim workspace from the RMA.

Finally, the internals of the actual module code are described as follows:

- the entry points
- how the entry points are called
- how automatic command and SWI decoding is achieved etc.

*MODULES

Syntax: *MODULES

*MODULES lists all the system and relocatable modules which are currently present in the machine. System modules are stored in ROM, but may still be *UNPLUGged, or replaced by RAM-based modules. The names listed by this command are the module titles which are supplied to other commands, eg *RMKILL. This command also lists the base addresses and workspace areas of the modules.

The command *HELP MODULES provides information about the version numbers and creation dates of the currently installed modules.

*RMCLEAR

Syntax: *RMCLEAR

*RMCLEAR deletes all relocatable modules currently present in the RMA that it can and frees the work space that they have been assigned. System modules cannot be cleared in this way.

*RMKILL

Syntax: *RMKILL <module title>

*RMKILL deletes just the module whose name is given and releases its workspace. System modules may be removed by using this command. They are 'removed' until the next hard break or *RMREINIT.

*RMLOAD

Syntax: *RMLOAD <filename> [<module init string>]

*RMLOAD loads and initialises the relocatable module whose filename is given. The module will then declare itself on *HELP and any of its commands will be available for use.

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

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

*RMREINIT

Syntax: *RMREINIT <module title> [<module init string>]

*RMREINIT reinitialises the relocatable module, which is named and must be present in the machine. The module is returned to the same default state as when it has just been loaded. This command may be used to restore a module which has been *UNPLUGged or *RMKILLed.

*RMRUN

Syntax: *RMRUN <filename> [<module init string>]

*RMRUN runs the specified relocatable module. Such a module will typically contain an application, such as a language or word processor.

*RMTIDY

Syntax: *RMTIDY

*RMTIDY compacts and garbage collects the workspace used by the relocatable modules so that all free space is collected into a consecutive chunk of memory. This may affect the operation of certain modules such as causing files to be closed and sound to be interrupted.

*UNPLUG

Syntax: *UNPLUG [<module title>]

This command prevents the initialisation of a system (ROM-based) module. Once unplugged, a module can't be accessed until it is *RMREINITed. Even switching the machine off and on again won't affect the module's unplugged state. If the command is issued without an argument, a list of the modules currently unplugged is displayed.

MODULE SWI CALLS

This section describes the functions performed by the OS_Module call, and other secondary SWIs related to module. To understand these calls fully, you may also have to read the section on module entry points and header formats.

OS_Module &1E (30) – Perform a module operation

On entry: R0 = action code
R1... as below

On exit: R1... as below

The module handler provides OS_Module to manipulate modules. When loading a module it checks that the code has a load address of the form &FFFFFAxx, ie the file must be a stamped one with the type FFA. This prevents text files, for example, being loaded as modules.

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

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

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

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

R0 = 0 Run

On entry: R1 = pointer to filename plus optional parameters

On exit: Does not return if module has a start entry

This call is equivalent to loading then entering the module. If the module can be started as an application, it will be, and so the call will not return. Possible errors are `File not found`, `No room in RMA`, `Not a module`, `Duplicate module refused to die`, and `Module refuses to initialise`.

R0 = 1 Load

On entry: R1 = pointer to filename and optional parameters

On exit: V set on error

This instruction attempts to claim a block of the RMA and *LOADs the file if it has the correct file type. Then it attempts to kill any existing module of the same name. It sets the private workspace word to 0, calls the module through its initialise address and links it to the end of the module list, or replaces the old module of the same name.

The filename should be terminated suitably for OS_File. The terminator can be space, in which case there can be a parameter string after the filename to pass to the module initialisation. Possible errors are `File not found`, `No room in RMA`, `Not a module`, `Duplicate module refused to die`, and `Module refuses to initialise`.

R0 = 2 Enter

On entry: R1 = pointer to module name
R2 = pointer to parameters

On exit: normally doesn't return

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

R0 = 3 ReInit

On entry: R1 = pointer to module name plus any parameters for initialisation

On exit: V set on error

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

R0 = 4 Delete

On entry: R1 = pointer to name

On exit: V set on error

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

R0 = 5 Describe RMA

On entry: –

On exit: R2 = size of largest block claimable
R3 = total amount free in RMA

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

R0 = 6 Claim

On entry: R3 = contains the size wanted

On exit: R2 = pointer to claimed block
V set if block could not be allocated

This calls the heap manager to claim workspace in the RMA. If it fails and application workspace is not currently being used then it will attempt to reallocate this memory and retry. It returns with V set if it is still unsuccessful. This call is useful for claiming workspace during the module's initialisation, but may also be used from other module entries. The possible error is `No room in RMA`.

R0 = 7 Free

On entry: R2 = pointer to block

On exit: V set if block could not be freed

This calls the heap manager to free a block of workspace claimed from the RMA. The possible error is `Not a heap block`.

R0 = 8 Tidy

On entry: –

On exit: V set on error

This gives each module in turn, from the end of the list and working backwards, a non-fatal finalisation call. After all the modules have been called, it collects the RMA together into one large unfragmented block and reinitialises the modules again. Any private words containing pointers to workspace blocks in the RMA are relocated. Errors are generated if modules fail to die or reinitialise.

R0 = 9 Clear

On entry: –

On exit: V set on error

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

R0 = 10 Insert in-store module

On entry: R1 = pointer to start of block of memory

On exit: V set on error

This takes a pointer to a block of memory and links it into the module chain, without moving it. No checks are made on the validity of a module. Possible errors are `Duplicate module refuses to die` and `Module refuses to initialise`.

– *Note*: for future compatibility, the word immediately before the module start (ie at address R1 – 4) should contain the length of the module in bytes.

R0 = 11 Make RMA module from store area

On entry: R1 = pointer to start of module
R2 = length of module

On exit: V set on error

This takes a pointer to a block of memory, kills any duplicate module, copies the block into the RMA, initialises it and links it into the module chain. Possible errors are `Duplicate module refuses to die`, `No room in RMA` and `Module refuses to initialise`.

R0 = 12 Extract module information

On entry: R1 = module pointer or 0 for first call

On exit: R1 = module base or 0 if no more modules
R2 = private word

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

The information returned is exactly that printed by the *MODULES command.

Note that on versions of the OS after 0.40, this call may change dramatically in its input and output parameters. It is not recommended that you build it into any production programs at this time.

R0 = 13 Extend block

On entry: R2 = pointer to workspace block
R3 = amount to change block by

On exit: R2 = pointer to the allocated block
V set if block could not be altered

This allows modules to extend workspace blocks claimed in the RMA. It calls OS_Heap with the appropriate descriptor and attempts to enlarge the RMA if this fails. The possible error is `Can't extend block`.

OS_ServiceCall &30 (48)

On entry: R1 = service number
R0, R2 – R4 depend on R1

On exit: –

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

OS_Byte &8F (143) – Issue module service call

On entry: R1 = service type
R2 = argument for service

On exit: R2 may contain a return argument

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

WRITING A MODULE

This section contains the information you will need to write a relocatable module. It explains the module header fields, and how the code at each module entry point should behave.

Workspace

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

The system works on the assumption that the private word is a pointer to workspace claimed in the RMA. It therefore provides suitable default actions on that basis. For example, if a module has no finalisation entry, the system will attempt to free any workspace claimed using this pointer, when finalisation would otherwise be called.

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

- *Note*: workspace allocated through OS_Module will always lie on an address &XXXXXX4. This ensures that code poked into that area will execute as many consecutive (fast) cycles as possible. This is important for some very time-critical software, eg sound voice generators and FIQ handlers.

Errors in module code

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

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

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

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

Module header format

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

Offset	**Contains offset to**
&00	Start code
&04	Initialisation code
&08	Finalisation code
&0C	Service call handler
&10	Title string
&14	Help string
&18	Help and command keyword table
&1C	SWI chunk base number (optional)
&20	SWI handler code (optional)
&24	SWI decoding table (optional)
&28	SWI decoding code (optional)

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

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

Full details are given below.

Start Code

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

On entry: Entered in user mode with interrupts enabled.
R12 = pointer to the private word

On exit: Exit using OS_Exit, or by starting another application without setting up an exit handler.

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

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

Initialisation code

The Initialisation code is used by OS_Module with Run, Load, ReInit and Tidy reason codes.

On entry: Entered in Supervisor mode.
R13 = supervisor stack
Location pointed to by R12 <> 0 implies reinitialisation
R11 is always 0 or base if loaded from a podule
R10 points at the environment string (ie command tail)

On exit: Must preserve processor mode and interrupt state
Must preserve R7 – R11 and R13
R0 – R6, R12, R14 and the flags (except V of course) can be corrupted

Use the link register passed in R14 to return:

```
MOV PC,R14
```

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

This code is called when the module is loaded and also after the RMA has been tidied (OS_Module with Tidy reason code). It is defined that the module will not be called via any other entry point until this entry point has been called. Thus the initialisation code is expected to set up enough information to make all other entry points safe.

An offset of zero means that the module does not need any initialisation. The system does not provide any default actions.

If the module is being re-entered after a OS_Module 'tidy', the private word may contain a non-zero value. This is the contents of the private word before the finalisation, relocated (if necessary) by the system.

Typical actions are claiming workspace (via OS_Module) and storing the workspace pointer in the private word. Other actions may include linking onto vectors, declaring the module as a filing system, etc.

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

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

Finalisation code

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

On entry:	R12 = private word pointer
	R13 = supervisor stack
	R10 = fatality indication

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

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

The module should not enable interrupts if they are off unless it can cope with being entered via the service entry at that point.

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

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

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

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

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

If the module generates an error on finalisation, then it remains in the RMA, and is assumed to still be initialised.

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

Service call handler

The service call handler is used when a service call is issued or via an OS_Byte &8F (143) or OS_ServiceCall (see above for these calls)

On entry: May be entered in supervisor or interrupt mode depending on the service
Interrupts may be enabled or disabled
R1 = service number
R12 = private word pointer
R13 = a full, descending stack

On exit: R1 can be set to zero if the service is being claimed
R2 can be altered to pass back a result
R12 may be corrupted
Other constraints depend on the service

Use the value of R14 passed to return

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

An offset of zero means that the module is not interested in any service calls.

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

R1 = &00 Service call claimed

On entry: –

On exit: R1 = 0

This is the return code used to indicate that the module is claiming the service. A module is never called with this reason code. Note that there are some services which you should never call.

R1 = &04 Unknown command

On entry: R0 = pointer to command

On exit: R1 = 0 to claim the call the command
R0 = 0 for no error, else error pointer
R1 preserved to pass the call on

If you claim the call and execute the command successfully you should set R1 to 0. If an error occurs during execution then you should return with the pointer to the error buffer in R0.

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

R1 = &06 Error

On entry: R0 = pointer to error

On exit: –

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

R1 = &07 Unknown OS_Byte

On entry: R2 = OS_Byte number
R3 = first parameter
R4 = second parameter

On exit: R1 = 0 to claim the call, preserved otherwise
Errors cannot be returned

If the OS_Byte number is one of yours, you should execute it and claim the call by setting R1 to zero.

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

R1 = &08 Unknown OS_Word

On entry: R2 = OS_Word number
R3 = OS_Word parameter

On exit: R1 = 0 to claim the call, preserved otherwise
Errors cannot be returned

The same action applied as the OS_Byte entry.

*R1 = &09 *Help*

On entry: R0 = pointer to command

On exit: –

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

R1 = &0B Release FIQ

On entry: –

On exit: –

This is issued immediately after the FIQ handler is released. It must only be issued from foreground tasks. See the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS** for information on the hardware vectors.

R1 = &0C Claim FIQ

On entry: –

On exit: –

This is issued before the FIQ handler is claimed. It must only be issued from foreground tasks. See the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS** for information on the hardware vectors.

R1 = &11 Memory

On entry: R2 = active module pointer

On exit: R1 = 0 if claimed

This is issued when the contents-addressable memory (CAM) in the memory controller is about to be remapped, which alters the memory map of the machine. You should claim this call if you don't wish the remapping to take place.

If your module is the current application, and is using the application workspace, it should claim this call. This is because the remapping of the CAM generally involves moving memory from the application workspace to the RMA (so a module can be loaded, for example).

R2 on entry contains a pointer to the module currently active. BASIC checks this to see if it lies within its code. If it does (ie BASIC is currently active), BASIC claims the call.

R1 = &12 StartUpFS

On entry: R2 = filing system number

On exit: –

This is issued when a new filing system is about to be started, ie the user has typed a command such as *ADFS, *NET etc. It shouldn't be claimed.

R1 = &27 Reset

On entry: –

On exit: –

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

*R1 = &28 Unknown *CONFIGURE*

On entry: R0 = pointer to command tail, or 0 if none given

On exit: R1 = 0 if configure option recognised and no error
R0 <= 0 for no error
R0 = small integer for errors described below
R0 = error pointer for other errors (post 0.40 OS only)
Registers preserved if no command tail or not recognised

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

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

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

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

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

R1 = &29 Unknown *STATUS

On entry: R0 = pointer to command tail, or 0 if none given

On exit: R1 = 0 if status option recognised and no error
Registers preserved if no command tail or not recognised

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

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

Note that it is also possible to trap unknown *STATUS commands through the module's command table. Only one of these mechanisms should be used.

&2A – Application about to start

On entry: –

On exit: R1 = 0 to prevent application from starting

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

&40 – Filing system re-initialise

On entry: –

On exit: –

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

&42 – Lookup file type

On entry: R2 = file type (bits 0 – 11)

On exit: R1 = 0 if you know the file type:
R2 = first four characters
R3 = last four characters
R1 preserved if you don't know the file type

This call is passed round when FileSwitch would like to convert a twelve-bit file type into a textual name. If the file type passed in R2 is known to you, you should return with R1=0, and R2, R3 containing the eight characters in the name. This might be loaded as follows:

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

If no-one claims the call, FileSwitch will convert the number into a three-digit hex value padded with spaces.

&43 – International service

On entry: R2 = sub reason code
R3 – R5 depend on R2

On exit: R4 – 5 depend on R2 on entry

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

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

R2	Service required
0	Convert country name to country number
1	Convert alphabet name to alphabet number
2	Convert country number to country name
3	Convert alphabet number to alphabet name
4	Convert country number to alphabet number
5	Define range of characters

For reason codes 0 and 1, R3 contains a pointer to a null-terminated string. If the module recognises the country/alphabet name in that string, it should set R4 to the appropriate number and claim the call.

For reason codes 2 and 3, R3 is the country/alphabet number. R4 is a pointer to a buffer, and R5 is the buffer's length. If the module recognises the number, it should convert it to the appropriate string in the buffer, return with R5 as the length of the string, and claim the call.

For reason code 4, R3 is the country number. If the module recognises this number, it should set R4 to the alphabet number for that country, and claim the call.

For reason code 5, R3 contains the alphabet number; R4 and R5 give the inclusive range of ASCII codes which should be defined from that alphabet (using VDU 23 sequences).

Here is a list of the currently-defined country codes (provided by the international module), and the alphabets they use:

Code	Country	Alphabet
0	Default	
1	UK	101
2	Master	100
3	Compact	100
4	Italy	101
5	Spain	101
6	France	101
7	Germany	101
8	Portugal	101
9	Esperanto	103
10	Greece	107

Here is a list of the alphabet codes currently defined, provided by the international module:

Code	Alphabet
100	Bfont
101	Latin1
102	Latin2
103	Latin3
104	Latin4
107	Greece

&44 – Keyboard handler

On entry: R2 = keyboard id: 0 for 'old style' keyboard, 1 for A300 – 400 series keyboard

On exit: Don't claim

This call is made on reset, when the OS has established which type of keyboard is present, and after an OS_InstallKeyHandler SWI. It is for the information of keyboard handler modules which need to know what sort of keyboard is present; it should not be claimed.

&45 – Pre-reset

On entry: –

On exit: –

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

&46

See the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS** for information on the hardware vectors.

Mode change

On entry: –

On exit: Don't claim

This call is made whenever a mode change has taken place. It is made for the benefit of modules which may want to re-read some VDU variables to keep a consistent view of the world. It should not be claimed; there is nothing a module can do to prevent the mode change from taking place.

Title string

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

On entry: N/A

On exit: N/A

This is the offset of a null-terminated string which is used to refer to the module when OS_Module is called. It should not contain spaces or control characters. If a title string is not required, then a title of spaces is used.

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

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

Help String

Used when *HELP prints information from the module.

On entry: N/A

On exit: N/A

This is the offset of a null-terminated string printed out by *HELP before any information from the module, eg *HELP MODULES, *HELP COMMANDS. It is advisable that this string is present to avoid confusion. The string must not contain any control characters (except Tab, which tabs to the next multiple of eight column) but may contain spaces.

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

XT Module name v.vv (DD MMM YYYY) ET

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

Help and command keyword table

Used when OSCLI, *STATUS, *CONFIGURE and *HELP wish to look for user-supplied keywords.

On entry: Dependent on use.

On exit: Use the given link register for normal exit.
Return V set and R0 = error pointer if anything goes wrong.

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

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

String to match, null terminated
(ALIGN to a word boundary)
Offset of code from module start
Information word
Offset of invalid syntax message from module start, null terminated
Offset of help text from module start, null terminated

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

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

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

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

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

Byte 1 works as follows. Each bit corresponds to one parameter (bit zero of the byte = the first parameter and so on). If the bit is set, the parameter is OS_GSTransed before being passed on to the module. If the bit is clear, the parameter is passed directly to the module. This is useful for commands which take filenames which might contain variable references.

The flags are as follows:

Bit 31 = 1	The match string is a filing system command and is therefore only matched after OSCLI has failed to find the command in any of the module tables as a 'normal' command. OSCLI only looks at filing system commands in the filing system currently active. Commands that need this flag set are, therefore, the filing system-specific ones such as *BYE, *LOGON, etc.

Bit 30 = 1	The string is to be matched by *STATUS and *CONFIGURE. The code in this case should scan the command tail and return a status string or set non-volatile memory as appropriate. The code is called with R0 set as follows:

R0 = 0 *CONFIGURE with no option has been received. The module prints a syntax string and return.

R0 = 1 *STATUS <keyword> has been issued. The module should print the currently configured status for this keyword.

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

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

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

Bit 29 = 1 *HELP offset refers to a piece of code to call for that keyword, instead of the offset of a text string. The code is called with the following entry conditions:

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

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

Other flags should be zero for upwards compatibility.

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

The help text is used by *HELP. If a keyword in the *HELP command tail matches the match string, then the help text is pretty-printed. A zero offset means no help text is to be printed. The string may contain carriage returns to force newlines. Tab (ASCII 9) is also a special character; it forces alignment to the next multiple of eight columns. Finally, ASCII 31 is a 'hard space', around which words lines will not be split.

SWI chunk number and SWI handler code

Used when an unknown SWI is performed.

On entry: Entered in SVC mode with interrupts disabled
R11 = SWI number modulo Chunk Size (ie 0 – 63)
R12 = private word pointer
R13 = supervisor stack

On exit: R10 – R12 may be corrupted
Interrupts should be enabled if SWI processing will take a long time (say > 20us) and the routine can cope with IRQs being enabled. The code to enable (and re-disable) IRQs is:

```
MOV    Rn, R14
TEQP   Rn, #IRQ_Bit  ;=2
```

R14 contains the flags of the SWI caller. Use MOVS PC,R14 to return, having altered R14 flags as appropriate (eg setting V for an error).

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

Base to base + (SWI chunk size – 1)

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

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

- Base is a multiple of the chunk size and has a 0 top byte
- Code offset is a multiple of four with the top six bits zero

See the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS** for more details on how to choose a chunk number.

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

```
.SWIentry
        LDR             R12, [R12]                  ; get workspace pointer
        CMP             R11, #(EndOfJumpTable - JumpTable)/4
        ADDCC           PC, PC, R11, LSL #2         ; dispatch if in range
        B               UnknownSWIerror             ; unknown SWI

.JumpTable
        B               MySWI_0
        B               MySWI_1
        ...............
        B               MySWI_n
.EndOfJumpTable

.UnknownSWIError
        ADR             R0, errMesg
        ORRS            PC, R14, #Overflow_Flag
.errMesg
        EQUD            &1E6                        ;Same as system message
        EQUS            "Unknown <module> operation"
        EQUB            0
```

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

The R14 given to the SWI code contains the flags of the SWI caller, except that V has been cleared. So, to return without updating the flags, use MOVS PC, R14. Otherwise alter the link register (for example by executing ORRS PC, R14, #Carry_Flag). The flags returned to the system are returned to the caller.

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

SWI decode table and SWI decode code

The following refers to the SWI decode code entry. These are used by OS_SWINumberTo/FromString.

On entry: R0 < 0 means a request to convert from text to number.
In this case, R1 points at the string to convert (terminated by a control character).

R0 >= 0 is the offset of the SWI within the module SWI chunk
R1 = pointer to a buffer
R2 = the offset within the buffer at which to place the text
R3 = limit of the buffer
R12 = private word pointer
R13 = supervisor stack

On exit:

R12 may be corrupted
R1 – R6 may be corrupted if R0 < 0 on entry
R4 – R6 may be corrupted if R0 >= 0 on entry

If R0 < 0 on entry, return R0 as the offset (0 – 63) in the module SWI chunk range if the string was recognized, and return R0 < 0 if not recognized.

If R0 >= 0 on entry, add appropriate text to the buffer and update R2 by the length of the text. A null terminator is added to the text by the system.

Return using MOV PC,R14.

These fields are used by the SWIs that convert between SWI numbers and SWI names. When converting from a number to a name, if a SWI in the chunk range of the module is to be converted, then the SWI decode table is inspected to see if it contains a name for the SWI. The table format is:

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

All names are null terminated. For example, the debugger's table is:

```
EQUS   "Debugger"
EQUB 0
EQUS   "Disassemble"
EQUB 0
EQUB 0
```

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

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

TIME AND DATE

This chapter describes the calls provided for dealing with time on the Archimedes. There are several medium-resolution timers (100 ticks per second), in addition to the real-time CMOS clock/calendar. Calls are provided to convert between various date formats.

The real-time clock is used by the filing system manager to time- and date-stamp files. Stamped files have the time and date encoded into (what is normally) their load and execution addresses. They also have a twelve-bit file type, which is used by FileSwitch to determine how a file should be loaded or executed. See the chapter **FILING SYSTEMS** for a discussion of file stamping and types.

CENTI-SECOND TIMERS

There are four timers which increment at a centi-second rate, ie 100 times a second. Two of these are read/write timers, which you can alter as well as interrogate. The other two are read-only timers.

The two read/write timers are accessed through four OS_Word calls. All of these calls take a pointer to a five-byte parameter block, which is used to hold the centi-second value. Whether the timer is read or written and which timer is accessed, is determined by the OS_Word reason code. The calls are as follows:

OS_Word &01 (1) – Read system clock

Parameter block size: 5

On entry: The parameter block is unused.

On exit: The parameter block contains the value of the system clock at the instant of the call:

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

The system clock is used by the BASIC pseudo-variable TIME. The clock is incremented every centi-second. The value of the clock is preserved over a soft break and set to zero after a hard break.

OS_Word &02 (2) – Write system clock

Parameter block size: 5

On entry: The parameter block contains the new value of the system clock.

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

On exit: The parameter block remains unchanged.

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

OS_Word &03 (3) – Read interval timer

Parameter block size: 5

On entry: The parameter block is unused.

On exit: The parameter block contains the value of the interval timer at the instant of the call:

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

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

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

OS_Word &04 (4) – Write interval timer

Parameter block size: 5

On entry: The parameter block contains the new value for the interval timer:

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

On exit: The parameter block remains unchanged.

This call resets the interval timer to a specified value.

OS_Byte &F3 (243) – Read/write timer switch state

On entry: R1 = &FF
R2 = 0

On exit: R1 = switch state

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

The third centi-second timer is read using a SWI. It cannot be written. The SWI is OS_ReadMonotonicTime.

OS_ReadMonotonicTime &42 (66)

On entry: –

On exit: R0 = time

OS_ReadMonotonicTime returns the number of centi-seconds since the machine was switched on. 'Monotonic' refers to the fact that this timer cannot be written to, and so provides a value which is always guaranteed to increase with time. It is used, for example, to time-stamp mouse events.

The final centi-second timer is actually an encoded version of the real-time clock's value. It gives the number of centi-seconds since 00:00:00 1st Jan 1900. See the section **The real-time clock/calendar** for details.

In addition to these accessible timers, a program may install itself on one of the event chains that are called under interrupts. The interval timer zero-crossing event was mentioned above. There is also a 100Hz event, which is called, if enabled, 100 times second. See the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS** for details of these events.

There are also some SWIs which can be used to install and remove a routine from a timer chain. These are independent of the event routines, but are used in a similar way. The routine can be called once after a given number of centi-seconds have elapsed, or repeatedly every 'n' centi-seconds. The SWIs are:

OS_CallAfter &3B (59)

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

On exit: –

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

OS_CallEvery &3C (60)

On entry: R0 = interval in centi-seconds
R1 = address to call
R2 = value of R12 to call code with

On exit: –

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

OS_RemoveTickerEvent &3D (61)

On entry: R0 = address
R1 = R12 value used in OS_CallEvery

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

VSYNC TIMERS

Another regular event that occurs is the VSYNC signal from the video circuitry. This interrupts the processor every time the electron beam which 'draws' the picture on the display reaches the bottom of the display area of the screen. This occurs 50 times a second in the UK for modes 0 – 20.

Several things happen on this interrupt. A single-byte counter which is accessible using an OS_Byte is decremented. Any routines on the VSYNC vector are called. Finally, any foreground program waiting for a VSYNC (having called OS_Byte 19) will resume.

See the chapter **THE VDU DRIVERS** for a description of OS_Byte &13. The single-byte VSYNC counter is accessed as described below:

OS_Byte &B0 (176) – 50Hz counter

On entry: R1 = 0 or value to write
R2 = 255 or 0

On exit: R1 = previous ticker value
R2 = value of the next location (input source)

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

THE REAL-TIME CLOCK/CALENDAR

The OS_Words described below are used to read and set the time and date held in the battery-backed clock. This device keeps time even when the machine is switched off. It is used by the BASIC TIME$ pseudo-variable, and the *TIME command. The filing system also uses it for file date-stamping.

A variety of formats are available, including a compact five-byte format, which encodes the time and date as the number of centi-seconds since 00:00:00 1st Jan 1900. A couple of SWIs are provided to convert from this format to a textual string.

OS_Word &0E (14) – Read CMOS clock

This provides four different read functions associated with the CMOS clock.

Read clock in string format

Parameter block size: 25

On entry: The first byte of the parameter block indicates the function:

R1+0=0

On exit: The parameter block contains a 24-byte character string in the form:

where

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

R1+24 contains a carriage return character (&0D).

Read clock in Binary Coded Decimal (BCD) format

Parameter block size: 7

On entry: The first byte of the parameter block indicates the function:

R1+0=1

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

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

Convert BCD clock value into string format

Parameter block size: 25

On entry: The first byte of the parameter block indicates the function, the following seven contain the BCD clock value to be converted:

Convert BCD clock value into string format

Parameter block size: 25

On entry:	The first byte of the parameter block indicates the function, the following seven contain the BCD clock value to be converted:

R1+0 = 2	
R1+1 = year	(00 – 99)
R1+2 = month	(01 – 12; 01 = January etc)
R1+3 = day of month	(01 – 31)
R1+4 = day of week	(01 – 07; 01 = Sunday etc)
R1+5 = hrs	(00 – 23)
R1+6 = mins	(00 – 59)
R1+7 = secs	(00 – 59)

On exit:	The parameter block contains the 24-byte clock string (as defined under option one, above).

Read real-time in 5-byte format

On entry:	The first byte of the parameter block indicates the function.

R1+0=3

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

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

OS_Word &0F (15) – Write CMOS clock

This call provides three write functions associated with the CMOS clock.

Change the time only

Parameter block size: 9

On entry: The parameter block contains the new time:

R1+0 = 8
R1+1 = ASCII code for first hour's digit
R1+2 = ASCII code for second hour's digit
R1+3 = 58 (ie ASCII code for :)
R1+4 = ASCII code for first minute's digit
R1+5 = ASCII code for second minute's digit
R1+6 = 58
R1+7 = ASCII code for first second's digit
R1+8 = ASCII code for second second's digit

On exit: The parameter block remains unchanged.

Change the date only

Parameter block size: 16

On entry: The parameter block contains the new date:

R1+0 = 15
R1+1 = ASCII code for first day character
R1+2 = ASCII code for second day character
R1+3 = ASCII code for third day character
R1+4 = 44 (ie ASCII code for ',')
R1+5 = ASCII code for first day digit
R1+6 = ASCII code for second day digit
R1+7 = 32 (ie ASCII code for space)
R1+8 = ASCII code for first month character

R1+9 = ASCII code for second month character
R1+10 = ASCII code for third month character
R1+11 = 32
R1+12 = ASCII code for first year digit
R1+13 = ASCII code for second year digit
R1+14 = ASCII code for third year digit
R1+15 = ASCII code for fourth year digit

On exit: The parameter block remains unchanged.

Change date and time

Parameter block size: 25

On entry: The parameter block contains the new time and date:

R1+0 = 24
R1+1 – R1+15 = date string (as in 2, above)
R1+16 = 46 (ie period)
R1+17 – R1+24 = time string (as in 1, above)

On exit: The parameter block remains unchanged.

OS_ConvertStandardDateAndTime &C0 (192)

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

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

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

See OS_ConvertDateAndTime for details of the format string.

OS_ConvertDateAndTime &C1 (193)

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

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

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

The format string is copied into the buffer for the result. However, whenever '%' appears in the format string, the next two characters are treated as a special field name which is replaced by a component of the current time. The field names, which may use upper or lower case, are:

Name	Value	Example
CS	Centi-seconds	99
SE	Seconds	59
MI	Minutes	05
12	Hours in 12 hour format	07
24	Hours in 24 hour format	23
AM	"AM" or "PM"	PM
PM	"AM" or "PM"	AM
WE	Weekday, in full	Thursday
W3	Weekday, in three characters	Thu
WN	Weekday, as a number	5
DY	Day of the month	01
ST	"st", "nd", "rd" or "th"	st
MO	Month name, in full	September
M3	Month name, in three characters	Sep
MN	Month as a number	09
CE	Century	19
YR	Year within century	87
WK	Week of the year, Mon to Sun	52
DN	Day of the year	364
0	Insert an ASCII 0 zero byte	
%	Insert a '%'	

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

NUMBER CONVERSIONS

This chapter describes the SWIs which perform conversions between binary and ASCII number formats. There are two SWIs which convert from ASCII to binary: OS_ReadUnsigned, which reads an unsigned (ie positive) number in any base between 2 and 36, and OS_EvaluateExpression, which can perform a complex analysis of a string or numeric expression.

In the other direction, there is the SWI OS_BinaryToDecimal, which converts a 32-bit signed binary number to decimal. For more general conversions, there is a family of calls which convert various length signed and unsigned binary numbers into binary, decimal and hexadecimal representations. Finally, a couple of calls are provided to convert Econet station numbers into ASCII.

A different type of conversion is from SWI number to name, and vice versa. A couple of SWIs are provided to perform these conversions. Any program which deals with SWI numbers (eg an assembler) should take advantage of these conversion routines, and allow the user to refer to SWIs by their names as well.

Finally, the routines OS_GSInit, OS_GSRead and OS_GSTrans are described. These are not ASCII to binary conversions as such, but they do perform a certain amount of translation on a string which contains special escape sequences. For example, they will look up a variable name enclosed in angled brackets and substitute the name for the variable's value.

ASCII TO BINARY CONVERSIONS

OS_ReadUnsigned &21 (33)

On entry: R0 = base in the range 2 – 36 (else 10 assumed)
R1 = pointer to string
R2 = maximum value (if R0 bit 29 set – see below)

On exit: R1 = pointer to terminator character or unaltered on error
R2 = value or 0 on error
V is set if the string is unsuitable and nothing is read

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

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

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

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

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

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

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

OS_EvaluateExpression &2D (45)

On entry: R0 = pointer to string
R1 = pointer to buffer
R2 = length of buffer

On exit: R1 indicates type returned
R2 = integer result or length of string in buffer
V is set if buffer overflowed

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

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

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

Any item which cannot immediately be treated as a string or a number is looked up as a variable. For example, in the expression FRED+1, FRED will be looked up as a variable. If the item isn't an extant variable name, the error `Unknown operand` is given.

String operands (enclosed in quotes) are OS_GSTransed. Hence you can obtain control codes using, for example, "<7>" for ASCII 7. Note however that vertical bar escape sequences (eg "|G" for ASCII 7) are not recognised.

The operators recognised by the expression evaluator are as follows:

Arithmetic operators

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

Logical operators

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

Bit operators

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

String operators

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

Conversions

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

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

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

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

BINARY TO ASCII CONVERSIONS

OS_BinaryToDecimal &28 (40)

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

On exit: R0 is preserved, unless V is set
R1 is preserved
R2 = number of characters given
V is set if buffer overflowed

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

The error `Buffer overflow` is given if the converted string is too long to fit in the buffer.

Binary to ASCII conversion SWIs &D0 – &EA (208 – 234)

This group of SWIs converts from binary to various ASCII string representations. They all have similar input and output conditions, as summarised below:

On entry: R0 = binary value to be converted
R1 = pointer to buffer for resulting string
R2 = size of buffer

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

&D0 – &D4

These calls convert from binary to a hexadecimal string. The calls are OS_ConvertHexN, where N is the number of ASCII digits in the output string. It is 1, 2, 4, 6 or 8. Only enough significant bits to perform the conversion are used, and leading zeros are always included, so the string is fixed length. No ampersand ('&') is included in the string.

&D5 – &D8

This group converts from binary into an unsigned decimal number. They are called OS_ConvertCardinalN, where N is 1, 2, 3 or 4, and refers to the number of bytes to be used from the input binary value. The string is not padded with leading zeros, and so is of variable length.

&D9 – &DC

This group converts from binary into a signed decimal number. They are called OS_ConvertIntegerN, where N is 1, 2, 3 or 4, and refers to the number of bytes to be used from the input binary value. The string is not padded with leading zeros, and so is of variable length. If the most significant bit of the N bytes used is set, the number is taken to be negative, and a leading '–' is produced.

&DD – &E0

This group converts from binary into an ASCII binary representation. They are called OS_ConvertBinaryN, where N is 1, 2, 3 or 4, and refers to the number of bytes to be used from the input binary value. The string is padded with leading zeros, and so is length N*8.

&E1 – &E4

This group converts from binary into an unsigned decimal number. They are called OS_ConvertSpacedCardinalN, where N is 1, 2, 3 or 4, and refers to the number of bytes to be used from the input binary value. The string is not padded with leading zeros, and so is of variable length. In addition, every three digits from the right, a space is inserted. Thus 65535 would be converted as 65 535.

&E5 – &E8

This group (&E5 – &E8) converts from binary into a signed decimal number. They are called OS_ConvertSpacedIntegerN, where N is 1, 2, 3 or 4, and refers to the number of bytes to be used from the input binary value. The string is not padded with leading zeros, and so is of variable length. If the most significant bit of the N bytes used is set, the number is taken to be negative, and a leading '–' is produced. In addition, every three digits from the right, a space is inserted. Thus –1000000 would be converted as –1 000 000.

OS_ConvertFixedNetStation and OS_ConvertNetStation

The final two conversions convert from an Econet station/network number pair into an ASCII version. The entry condition for these calls is slightly different from the rest, in that R0 points to two words in memory. The first word contains the station number and the second word contains the network number.

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

The second call is OS_ConvertNetStation. This performs the same conversion, but suppresses zeros and spaces wherever possible, to yield the shortest possible string.

SWI NUMBER AND NAME CONVERSIONS

These two SWIs enable you to convert from a standard format SWI name into the corresponding SWI number, and vice versa. A standard format name is [X]module_name. The optional X represents a bit 17-set (error-returning) SWI. The module part is a prefix appropriate to the module providing the SWI: OS for built-in operating system SWIs, Wimp for window manager SWIs etc. The final part is a descriptive name for the SWI, such as RemoveCursors.

OS_SWINumberToString &38 (56)

On entry: R0 = SWI number
R1 = pointer to buffer
R2 = buffer length

On exit: –

OS_SWINumberToString converts a SWI number to a SWI name.

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

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

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

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

OS_SWINumberFromString &39 (57)

On entry: R1 = pointer to a name terminated by an ASCII code <= 32

On exit: R0 = SWI number

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

The conversion is as follows:

- A leading X is checked for and stripped. If present, it will cause &20000 to be added to the number returned. (Bit 17 will be set.)

- System names are checked for. Note that the conversion of SWIs is not quite bidirectional: the name OS_WriteI+" " can be produced, but only OS_WriteI is recognized.

- Modules are scanned. If the module prefix matches the one given, and the suffix to the name is a number, then that number is added to the module's SWI 'chunk' base, and the sum returned. For example, Wimp_&23 returns &400E3, as the Wimp's chunk number is &400C0.

  If the suffix is a name, and this can be matched by the module, the appropriate number is returned. For example, Wimp_Poll returns &400C7.

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

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

STRING SCANNING ROUTINES

OS_GSInit &25 (37) – String input initialisation

OS_GSInit initialises registers for use by OS_GSRead.

On entry: R0 = pointer to string to translate
R2 = flags

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

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

See OS_GSRead below for a list of the conversions that are performed by the routines.

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

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

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

OS_GSRead &26 (38) – String input

This routine returns a character from a string which has previously been initialised by OS_GSInit.

On entry:	R0 from last OS_GSRead/OS_GSInit R2 from last OS_GSRead/OS_GSInit

On exit: R0 is updated
R1 = next translated character
R2 is updated
V set if bad string (eg mismatched quotation mark)
C set if end of string reached

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

- |c, etc, are converted into the relevant control codes, unless these were disabled by bit 30 of R2, which was set when OS_GSInit was called.
- an unsigned integer within angled brackets is evaluated using OS_ReadUnsigned and the corresponding ASCII character is returned, eg <2_1010> returns ASCII 10, <7> ASCII 7 etc.
- a variable name within angled brackets is replaced by the value held by the variable, or is ignored if the variable does not exist
- Anything else is returned unchanged.

A full list of ASCII codes and how to obtain them is given below. Of course, any ASCII code may be obtained by enclosing it in angled brackets as described above, and this may be easier to remember than the symbol encoding.

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

Note that you must use |< to prevent the '<' from being interpreted as the start of a number or variable name enclosed in angled brackets.

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

OS_GSTrans &27 (39) – General String Translation

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

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

On exit: R0 = pointer to terminator
R2 = number of characters or maxlen+1 if it overflowed
V is set if bad string (eg mismatched quotation mark)
C is set if buffer overflowed

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

OS_SubstituteArgs (&43) – Substitute command line arguments

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

The call is only available under OS 0.40 and above.

On entry: R0 = pointer to argument list
R1 = pointer to buffer for result string
R2 = length of buffer
R3 = pointer to template string
R4 = length of template string

On exit: R2 = number of characters in result string (inc. terminator)

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

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

The template string is copied into the result buffer character for character. However, when a '%' appears in the template string (even within quotation marks), it marks where an argument should be placed into the output buffer. The '%' is followed by a single digit from 0 to 9. %0 stands for the first argument in the argument list, and so on. %*n means all of the arguments from number n onwards. %% means a single '%'. Anything else following the '%' is not treated specially, ie both the '%' and the character are copied over.

The template string does not have a terminator; instead its length is given. At the end of the substitution, any arguments after the highest one mentioned in the template string are appended to the result string.

SPRITES

Sprites are graphics shapes which can be plotted in any of the graphics display modes. Sprite plotting is part of the VDU drivers, and is accessed through the SWI OS_SpriteOp

A system module, called SpriteUtils, provides a * command interface to so-called system sprites. These can be plotted, saved and loaded, merged with other sprite files, and defined from rectangular areas of the screen.

A sprite is a rectangular array of pixels. The pixels in the sprite can be set to any of the logical colours available. When a sprite is created, the numbers of bits per pixel is given. The sprite can only be plotted in modes which have the same number of bits per pixel.

The pattern of pixels that make up a sprite can be defined in two principal ways. It can be 'grabbed' from the screen, so that the sprite is just a copy of a section of the screen. Alternatively, a sprite can be defined using a sprite editor utility. Such a program is provided on the Archimedes Welcome disc.

In addition to the pixel pattern, a sprite can have a transparency mask and a palette table. The transparency mask is an array with the same dimensions as the sprite itself. Each pixel in the mask can be set to transparent or solid. When the sprite is plotted with the appropriate GCOL action, the transparency mask is used to determine which pixels of the sprite are actually plotted. Only those with the corresponding mask bit set to solid will be plotted.

The palette for the sprite can be used to set up the screen palette when the sprite is loaded, so that the colours are the same as when the sprite was created. This facility is used by the sprites which are created by the command *SCREENSAVE. (These are described below.)

There are two types of sprite: system and user. System sprites are held in the system sprite area. The size of this area is set by a *CONFIGURE option (SpriteSize). It is given in physical pages (8K or 32K bytes). System sprites are always referred to by their names, a sequence of up to 12 characters. This makes them ideal for handling through * commands. Certain system sprites, whose names happen to be the numbers 0 through 255, are also accessible through a VDU 23 command.

User sprites are stored in an area allocated by the user, for example the application workspace or RMA. They can only be accessed through the OS_SpriteOp call. These sprites can be referenced by name, or directly by a pointer to the sprite itself. This is the most efficient way of dealing with sprites.

SPRITE * COMMANDS

This section describes the * commands for dealing with system sprites.

*SCHOOSE

Syntax: *SCHOOSE <name>

*SCHOOSE selects a particular sprite for use in subsequent sprite plotting operations. The sprite is identified by giving the name with which it was created. If it was created using a VDU 23 command (see below), this will be a number in the range 0 to 255.

*SCOPY

Syntax: *SCOPY <name1> <name2>

*SCOPY makes a copy of sprite <name1> and names it <name2>.

*SCREENLOAD

Syntax: *SCREENLOAD <filename>

*SCREENLOAD plots a sprite directly from a file onto the screen, setting the palette to that held in the file.

If the current mode is the same as that held with the sprite, the sprite is plotted at the bottom left-hand corner of the graphics window. If the current mode is different, a mode change is performed and the sprite is plotted at the bottom left-hand corner of the screen.

***SCREENSAVE**

Syntax: *SCREENSAVE <filename>

*SCREENSAVE saves the current graphics window and palette information as a sprite file. It is given a type FF9, and lists as 'Sprite' in *EX and *INFO commands. The file contains one sprite, called 'screendump'.

***SDELETE**

Syntax: *SDELETE <name1> [<name2> ... <name3>]

*SDELETE deletes one or more sprites from memory.

***SFLIPX**

Syntax: *SFLIPX <name>

*SFLIPX reflects the named sprite about its X axis so it is upside down.

***SFLIPY**

Syntax: *SFLIPY <name>

*SFLIPY reflects the named sprite about its Y axis so it faces in the opposite direction.

***SGET**

Syntax: *SGET <name>

*SGET picks up a rectangular area of the screen, defined by the two previously-visited graphics points, and saves it as a sprite with the name given.

***SINFO**

Syntax: SINFO

*SINFO prints out the amount of sprite workspace currently reserved, the amount of free space in that workspace and the number of sprites defined.

***SLIST**

Syntax: *SLIST

*SLIST prints a list of the identifiers of all the sprites currently held in memory, or the message `No sprites defined`.

***SLOAD**

Syntax: *SLOAD <filename>

*SLOAD loads a file containing sprite definitions into memory so that these sprites may be edited, plotted, etc. If there is insufficient sprite workspace for all the sprites, a message is given and nothing is loaded. Any sprites which are in memory when the command is successfully given are lost.

***SMERGE**

Syntax: *SMERGE <filename>

*SMERGE merges the sprite definitions in the file named with those in memory. If the file contains a sprite with the same identifier as one in memory, the original version is overwritten.

***SNEW**

Syntax: *SNEW

*SNEW deletes all the sprite definitions currently in memory and so frees all the sprite workspace.

***SRENAME**

Syntax: *SRENAME <name1> <name2>

*SRENAME assigns the new name, <name2>, to the sprite whose current name is <name1>. A sprite name can contain any sequence of printable characters. Control characters cannot be used since they terminate the string.

***SSAVE**

Syntax: *SSAVE <filename>

*SSAVE saves all the sprites currently in memory to a file which can later be reloaded, or merged.

***CONFIGURE SpriteSize**

Syntax: *CONFIGURE SpriteSize <n>

This causes 'n' physical pages to be reserved for the sprite definition. The range of 'n' is 0 to 255. As with all *CONFIGUREs, the command does not come into effect until the next hard break.

SPRITE VDU COMMANDS

VDU23,27,m,n,0,0,0,0,0,0

VDU 23,27 is used to select or define a sprite.

m = 0	The command selects the sprite whose name is 'n'. It can, therefore, only be used for sprites whose names are 0 – 255. It is equivalent to OSCLI"SCHOOSE " + STR$n in BASIC.
m = 1	The command defines sprite 'n' to contain the contents of the rectangular area of the screen whose extents are determined by the current and previous graphics cursor positions. It is equivalent to OSCLI"SGET " + STR$n in BASIC.

These VDU commands are provided for compatibility with the BBC Micro's Graphics Extension ROM and Master Compact sprite ROM. In new code, the *SCHOOSE and *SGET commands, or the corresponding SWI OS_SpriteOp calls, should be used instead.

VDU 25,232 – 239,x;y; (PLOT 232 – 239,x,y) Plot a sprite

These calls plot the sprite currently selected. The co-ordinates, and the way in which the sprite is plotted depend on the plot code:

Plot	Meaning
232	Move by x,y
233	Plot sprite by x,y
234	Invert sprite-sized rectangle by x,y
235	Plot sprite mask in background colour by x,y
236	Move to x,y
237	Plot sprite at x,y
238	Invert sprite-sized rectangle at x,y
239	Plot sprite mask in background colour at x,y

It is quicker to use OS_SpriteOp to plot a particular sprite.

THE SPRITE SWI CALL

The most versatile way of dealing with sprites is through SWI OS_SpriteOp (&2E). This SWI is more powerful than the higher level commands since it may be used to define user sprite areas, in addition to the system one, and to manipulate sprites in those areas as well.

The full list of actions performed by SWI OS_SpriteOp is given below. In general:

On entry: R0 = reason code
R1...R6 depend on R0

On exit: depends on R0

The reason code, given in R0, falls into one of three distinct ranges, depending on what sort of sprite operation it is. They are as follows:

Range	Type of operation
0 – 7	High-level sprite operations
8 – 23	Sprite file and get/put operations
24 – X	Low-level sprite operations

X is the highest sprite operation code, which is 49 currently.

The first group covers screen save/load operations and deals with a new sprite area. The second group covers many of the operations made available on system sprites through the SpriteUtils module. The last group deals with manipulating individual sprites, such as setting their transparency masks, changing the sprite size, etc.

Now, in addition to the three groups of sprite operations described above, there are three ways in which sprites may be referenced, depending on whether they are 'system' or 'user' sprites, and whether you want to access them by name or address (user sprites only).

You tell the sprite system which type of access you require by adding a number to the basic range given above. If you add zero (ie use the base range), then system sprites are used – these are always accessed by name.

If you add 256 to the operation code, then user sprites will be accessed, and the access will be by name.

If you add 512, user sprites will again be accessed. This time though, instead of providing a pointer to the sprite name, you give a pointer to the sprite itself. This gives very efficient access, and is the fastest way of dealing with a sprite.

To use non-system sprites, you must claim some workspace from the sprite area (eg by DIMing space in BASIC or claiming it for the RMA if you are a module), then set up a sprite header at the start of this space. The sprite header format is shown in the section **Sprite internals**. (This follows later.)

The table below summarises how R1 and R2 are used according to the sprite group you are using (0 – 7, 8 – 23 or 24 – X), and the number you add to the basic operation code.

R0 = 0 + n

On entry: R1 is unused – sprites are accessed in the system area (operations 8 to X)
R2 = pointer to a sprite name (operations 24 to X)

These allow you to manipulate sprites in the system area. The sprite is identified by its name which is pointed to by R2. This applies to operation codes in the range 24 to X. The first and second groups of codes do not use R2 to access particular sprites.

R0 = 256 + n

On entry: R1 = pointer to user sprite area (operations 8 to X)
R2 = pointer to a name of user area sprite (operations 24 – X)

You can set up user sprite areas and manipulate sprites in these areas using these calls. Sprite operations in the range 8 to X use R1 as the pointer to the sprite area. Operations in the range 24 to X, which access particular sprites, use R2 to point to the sprite name.

R0 = 512 + n

On entry: R1 = pointer to user sprite area (operations 8 to X)
R2 = pointer to a sprite (operations 24 to X)

You can set up user sprite areas and manipulate sprites in these areas with these calls. Sprite operations in the range 8 to X use R1 as the pointer to the sprite area. Operations in the range 24 to X, which access particular sprites, use R2 to point to the sprite itself. As some sprite operations can change the address of a sprite (eg the add sprite mask call), you should re-discover the address using, for example, the select sprite operation, if you think the address has changed.

The full range of operations is given below. The value of the offset, 'n', within each range is given and the parameters should be set according to the range (ie sprite type) used:

n = 2 – Screen save

On entry: R2 = pointer to filename
R3 = palette flag

On exit: –

This saves the current graphics window as a sprite file. The file contains a single sprite called 'screendump'. If R3 is 0, no palette information is saved with the file; if it is 1, the current palette is saved. It is equivalent to *SCREENSAVE. All versions of this call have the same effect.

n = 3 – Screen load

On entry: R2 = pointer to filename

On exit: –

This plots a sprite directly from a file to the screen. It changes mode if necessary and sets the palette to the setting held in the file. The sprite is plotted at the bottom left of the graphics window. After a mode change, this is the bottom left-hand corner of the screen. It is equivalent to *SCREENLOAD. All versions of this call have the same effect.

Calls described from this point use the operation code to determine whether R1 is unused (0+n so system sprites are used), or contains a pointer to a user area (256+n and 512+n).

n = 8 – Read area control block

On entry: R1 = pointer to control block of sprite area

On exit: R2 = total size of sprite area in bytes
R3 = number of sprites in area
R4 = byte offset to the first sprite
R5 = byte offset to the first free word

This returns all the information contained in the control block of a sprite area.

n = 9 – Clear sprite area

On entry: R1 = pointer to control block of sprite area

On exit: –

This reinitialises a sprite area. For system sprites, it is equivalent to *SNEW.

n = 10 – Load sprite file

On entry: R1 = pointer to control block of sprite area
R2 = pointer to filename

On exit: –

This loads the sprite definitions contained in the file into the sprite area, overwriting any definitions stored there already. It is equivalent to *SLOAD for the system area.

n = 11 – Merge sprite file

On entry: R1 = pointer to control block of sprite area
R2 = pointer to filename

On exit: –

This merges the sprite definitions contained in the file with those in the sprite area. For system area sprite, it is equivalent to *SMERGE.

n = 12 – Save sprite file

On entry: R1 = pointer to control block of sprite area
R2 = pointer to filename

On exit: –

This saves the contents of a sprite area to a file. It is equivalent to *SSAVE when used with the system area.

n = 13 – Return name

On entry: R1 = pointer to control block of sprite area
R2 = pointer to buffer
R3 = maximum buffer length
R4 = sprite number (position in workspace)

On exit: R3 = string length

This returns the name of the sprite whose position in the workspace (eg 3 for the third sprite) is given in R4. The name is placed in the buffer pointed to by R2 and the string length is returned in R3.

n = 14 – Get sprite

On entry: R1 = pointer to control block of sprite area
R2 = pointer to sprite name
R3 = palette flag

On exit: R2 = address of sprite (if sprite is in a user sprite area)

This defines the sprite identified to be the current contents of an area of the screen which is delimited by the current and old cursor positions. The palette flag in R3 has the following effect:

Value	**Meaning**
0	Exclude palette data
1	Include palette data

For the system sprite area, this is equivalent to *SGET.

n = 15 – Create sprite

On entry: R1 = pointer to control block of sprite area
R2 = pointer to sprite name
R3 = palette flag
R4 = width in pixels
R5 = height in pixels
R6 = mode number

On exit: R2 = address of sprite (if sprite is in a user sprite area)

This creates a blank sprite of a given size. The palette flag in R3 has the following effect:

Value	**Meaning**
0	Exclude data
1	Include data

n = 16 – Get sprite from user co-ordinates

On entry: R1 = pointer to control block of sprite area
R2 = pointer to sprite name
R3 = palette flag
R4 = left-hand edge (external co-ordinates)
R5 = bottom edge (external co-ordinates)
R6 = right-hand edge (external co-ordinates)
R7 = top edge (external co-ordinates)

On exit: –

This picks up an area of the screen, which is delimited by the co-ordinates supplied, as a sprite. The palette flag in R3 has the following effect:

Value	Meaning
0	Exclude data
1	Include data

Calls described from this point on use the reason code to determine how R1 and R2 are used.

– If R0=0+n, then R1 isn't used (the system sprite area is assumed) and R2 contains a pointer to the sprite name.

– If R0=256+n then the user sprite area addressed by R1 is used, and R2 contains a pointer to a sprite name.

– If R0=512+n, then the user sprite area addressed by R1 is used, and R2 contains the address of the sprite to be accessed.

The phrase 'sprite pointer' should, therefore, be taken to mean whichever type is appropriate for the reason code.

n = 24 – Select sprite

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit: R2 = address of sprite (if sprite is in a user sprite area)

If R0=0+24, this selects a particular sprite for subsequent plotting by PLOT &ED etc. It is equivalent to *SCHOOSE for system sprites.

If R0=256+24 or 512+24, this returns the absolute address of the sprite in R2.

n = 25 – Delete sprite

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit: –

This deletes the definition of a particular sprite; it is equivalent to *SDELETE for system mode sprites.

n = 26 – Rename sprite

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = pointer to new name

On exit: –

This changes the name of a sprite; it is equivalent to *SRENAME or the system sprite area. An error is produced if a sprite of the new name already exists.

n = 27 – Copy sprite

On entry:	R1 = pointer to control block of sprite area R2 = source sprite pointer R3 = pointer to new sprite name
On exit:	–

This copies a sprite within a sprite area, giving the copy a new name. An error is produced if there is a name clash between the copy and an existing sprite. It is equivalent to *SCOPY for system sprites.

n = 28 – Put sprite

On entry:	R1 = pointer to control block of sprite area
	R2 = sprite pointer
	R5 = GCOL action

On exit:	–

This plots the sprite identified at the graphics cursor position given using the GCOL action specified.

n = 29 – Create mask

On entry:	R1 = pointer to control block of sprite area R2 = sprite pointer
On exit:	–

This creates a mask for the given sprite with all pixels set to be solid, so the whole sprite is plotted.

n = 30 – Remove mask

On entry:	R1 = pointer to control block of sprite area R2 = sprite pointer

On exit: –

This removes the mask definition for a given sprite.

n = 31 – Insert row

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = row number

On exit: –

This inserts a row at the position identified, shifting all rows above it up one. All pixels in the new row are set to colour zero. Rows are numbered from the bottom upwards with the bottom row being number zero.

n = 32 – Delete row

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = row number

On exit: –

This deletes the row given, shifting all rows above it down one.

n = 33 – Flip about x-axis

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit: –

This takes the sprite identified and reflects it about the x-axis so that it is upside down. Thus, its previous top row becomes the bottom row, etc.

n = 34 – Put sprite at user co-ordinates

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x co-ordinate (external co-ordinates)
R4 = y co-ordinate (external co-ordinates)
R5 = GCOL action

On exit: –

This plots a sprite at the co-ordinates supplied, using the plot action supplied. See the section on VDU 18 (GCOL) for details of the various plot actions, how to use the sprite mask, etc.

n = 40 – Read sprite size

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit: R3 = width in pixels
R4 = height in pixels
R5 = mask status
R6 = screen mode in which the sprite was defined

This returns information about the sprite, giving its width and height in pixels, the screen mode in which the sprite was defined and its mask status which is given in R5 as follows:

Value	**Meaning**
0	No mask
1	Mask

n = 41 – Read pixel colour

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x co-ordinate
R4 = y co-ordinate

On exit: R5 = colour
R6 = tint (ignored for non-256-colour modes)

Given a pair of x and y co-ordinates in R3 and R4 (in pixels from the bottom left of the sprite definition), this SWI option returns the current colour of the pixel at that position in R5, and the tint in R6 if applicable.

n = 42 – Write pixel colour

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x co-ordinate
R4 = y co-ordinate
R5 = colour
R6 = tint (ignored for non-256-colour modes)

On exit: –

Given in R3 and R4, a pair of x and y co-ordinates (in pixels from the bottom left of the sprite definition), and a colour in R5 (and possibly a tint value in R6), this SWI option sets the pixel at the position given to that colour.

n = 43 – Read pixel mask

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x co-ordinate
R4 = y co-ordinate

On exit: R5 = mask status

Given in R3 and R4, a pair of x and y co-ordinates in pixels from the bottom left of the sprite definition, this SWI option returns the state for its mask at the position identified as follows:

R5 = 0	Transparent
R5 = 1	Solid

n = 44 – Write pixel mask

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x co-ordinate
R4 = y co-ordinate
R5 = mask status

On exit: –

Given in R3 and R4, a pair of x and y co-ordinates in pixels from the bottom left of the sprite definition, this SWI option sets the pixel at the position identified in its mask definition to be either transparent or solid as follows:

R5 = 0	Set pixel to be transparent
R5 = 1	Set pixel to be solid

n = 45 – Insert column

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = column number

On exit: –

This inserts a column at the position identified, shifting all columns after it one place to the right. All pixels in this new column will be set to colour zero. Columns are numbered from the left with the left-hand one being number zero.

n = 46 – Delete column

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = column number

On exit: –

This deletes the column given, shifting all columns after it one place to the left. Columns are numbered from the left with the left-hand one being number zero.

n = 47 – Flip about y-axis

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit: –

This takes the sprite identified and reflects it about the y-axis so that it is facing in the opposite direction, ie the left-most column becomes the right-most one, etc.

n = 48 – Plot sprite mask

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer

On exit: –

This plots in the background colour and action through a sprite mask at the graphics cursor position. That is, all 1 bits in the mask are plotted in the backgound colour and action, and all 0 bits are ignored. If the sprite has no mask, a solid rectangle the same size as the sprite is drawn in the current background colour and action (as if there was a mask and it was all ones).

n = 49 – Plot mask at user co-ordinates

On entry: R1 = pointer to control block of sprite area
R2 = sprite pointer
R3 = x co-ordinate (external co-ordinates)
R4 = y co-ordinate (external co-ordinates)

On exit: –

This plots in the background colour and action through a sprite mask at the co-ordinates supplied.

SPRITE INTERNALS

The format of a sprite area is as follows:

Control Block	Extension Area (Optional)	Sprite	Sprite	Free Space

The sprite area control block contains the following:

Word	**Contents**
1	Byte offset to last byte+1 (ie total size of sprite area)
2	Number of sprites in area
3	Byte offset to first sprite
4	Byte offset to first free word
5 ...	Extension words

The above offsets are relative to the start of the sprite area control block.

So, to build a user sprite area from BASIC, you might use:

```
spriteLen = &4000         : REM 16K sprite area
DIM mySprites spriteLen
!mySprites   = spriteLen : REM Init length word
mySprites!4  = 0         : REM Init sprite count
mySprites!8  = 16        : REM Init first sprite pointer
mySprites!12 = 16        : REM Init free space pointer
```

The format of file created by *SCREENSAVE command is the same as a sprite area but without the first word of data (which would be the length). Note that all offsets are relative to this non-existent word.

The format of a sprite is as follows:

Control Block	Palette Area (Optional)	Sprite Image	Plotting Mask (Optional)

The Sprite Control Block contains the following:

Word	**Content**
1	Offset to next sprite
2 – 4	Sprite name, up to 12 characters with trailing nulls
5	Width in words –1
6	Height in scan lines –1
7	First bit used (left end of row)
8	Last bit used (right end of row)
9	Offset to sprite image
10	Offset to transparency mask or offset to sprite image if no mask
11	Mode sprite was defined in
12 ...	Palette data (optional)

THE WINDOW MANAGER

INTRODUCTION

– *Note*: this chapter documents the window manager version 0.18.

The window manager is designed to simplify the task of producing application programs to run under a 'WIMP' (Windows, Icons, Menus and Pointer) environment. The manager itself is often referred to as the Wimp.

The window manager cooperates with the application in keeping the screen display correct by telling the application when something needs to be redrawn. Thus, the application needs to make as few intelligent decisions as possible. It merely has to respond appropriately to the messages it receives from the window manager, in addition to performing its own processing (using the routines supplied to perform window operations).

The window manager also provides a facility for 'writeable icons', which are elements inside a window containing text. These elements can be automatically updated by the window manager when the user clicks on them and presses keys. This feature simplifies the job of obtaining text input from the user.

Layout of windows

The basic idea is that windows consist of a visible work area, in which the application can draw graphics, and a surrounding 'system' area, comprising a title bar, scroll bar indicators and so on. The application is not allowed to draw directly in this area. The visible work area provides a window into a larger region, called the work area extent. You can imagine the work area extent to be the complete document you are working with, and the visible area a window into this.

There are, therefore, two sets of co-ordinates to deal with when setting up a window. The visible work area co-ordinates determine where the window will appear on the screen and its size. These are given in terms of graphics co-ordinates, with the origin in its default position at the bottom left of the screen.

Then, there are the work area extent co-ordinates. These give the minimum and maximum x and y co-ordinate of the whole document. The extent is specified when a window is created, but can be altered using the Wimp_SetExtent call.

Between the extent co-ordinates and the visible region co-ordinates is a final pair which join the two together. These are the scroll offsets. These indicate which part of the extent is shown by the visible region. The scroll offsets give the co-ordinates of a pixel in the work area which is displayed at the top left-hand corner of the visible window. Say the visible region shows the very top left of the window extent. Then the x scroll position would be 'extent x min', and the y scroll position would be 'extent y max'.

It is common to define the extent work area such that its origin (0,0) is at the top left of the document. This means that all x scroll offsets are positive (as you can only ever be to the right of the extent origin), and all y offsets are zero or negative (as you can only ever be on or below the extent origin).

To summarise, let's consider which part of the workspace extent will be visible, and where it will appear on the screen, for a typical set of co-ordinates.

The following give the total document size:

extent x min = 0
extent y min = –5000
extent x max = 1000
extent y max = 0

The following give the window position and its size:

visible x min = 200
visible y min = 100
visible x max = 500
visible y max = 400

The following show which part of the extent is shown:

scroll x = 250
scroll y = –400

visible extent min x = scroll x = 250
visible extent min y = scroll y – height = –700
visible extent max x = scroll x + width = 550
visible extent max y = scroll y = –400

So, on the screen at co-ordinates (200,100) – (500,400) would be a 300 pixel-square window showing the work area region (250,–700) – (550,–400). Moreover, the scroll 'sausages' drawn by the system have a length proportional to the area that the window displays. The horizontal sausage would therefore occupy about 300/1000 = 0.3 of horizontal scroll bar, and the vertical one would occupy 300/5000 = 0.06 of the scroll bar.

Redrawing windows

As mentioned above, the Wimp and the program co-operate to ensure that the windows on the screen remain up to date. The Wimp can't do all of the work, as it does not always know what the contents of a window should be.

The Wimp requests a program to re-draw all or part of one of its windows by passing back a 'reason code' from the routine Wimp_Poll. This is the central SWI to any Wimp-based program, and must be called in the program's inner loop to ensure that all events which the program should know about (eg mouse clicks) are reported by the Wimp.

When the application receives the reason code Redraw_Window_Request from Wimp_Poll, it should enter a loop of the following form:

```
Call Wimp_RedrawWindow for window, returns 'flag'
WHILE 'flag' do
        Redraw contents of the appropriate window
        CALL Wimp_GetRectangle, returns 'flag'
ENDWHILE
Return to main polling loop
```

When a window has to be redrawn, often only part of it needs to be updated. The Wimp splits this area into a series of non-overlapping rectangles. The WHILE loop above is used to obtain all the rectangles so that they can be re-drawn. The Wimp automatically sets the graphics clipping window to the rectangle to be redrawn. The application can take a simplistic view, and redraw its whole window contents each time round the loop (relying on the graphics window to clip the unwanted parts out). Alternatively, and much more efficiently, it can inspect the graphics window co-ordinates and only draw the contents of that particular rectangular region.

The areas to be redrawn are automatically cleared (to the background colour) by the Wimp. The application must determine what part of the workspace area is to be redrawn using the top-left window extent co-ordinates and the current scroll values.

Updating windows

When an application wants to update a window's contents, it must NOT simply update the appropriate area of the screen. This is because the application does not know which other windows overlap the one to be updated, so it could over-write their contents. As with all window operations, it must be done with the Wimp's co-operation. There are two possible approaches:

– Call Wimp_ForceRedraw so Wimp issues a Redraw_Window_Request

– Call Wimp_UpdateWindow, and perform appropriate operations.

In both cases, you provide the window handle and the co-ordinates of the rectangular area to be updated (relative to the window origin). The Wimp works out which areas of this rectangle are visible, and marks them as invalid. If you use the first method, the Wimp will subsequently return a Redraw_Window_Request from Wimp_Poll, which you should respond to as described above. In the second case, a list of rectangles to be redrawn is returned immediately.

When Wimp_ForceRedraw is used, the Wimp clears the update area automatically. This should therefore be used when a permanent change has occurred in the window's contents, eg a paragraph has been reformatted in an editor. When you call Wimp_UpdateWindow, no such clearing takes place. This makes this call more suitable for temporary changes to the window, eg 'dragging' objects, or 'rubber-banding' in graphics programs.

It is simpler to use Wimp_ForceRedraw since, once it has been called, the application just returns to the central loop, from where the Redraw_Window_Request will be received. The code to handle this must already be present for the program to work at all. On the other hand, the second method is much quicker as the re-drawing is performed immediately.

Mouse buttons

The Wimp system works with a 3-button mouse, and since it is important that all applications use the mouse in as consistent a manner as possible, it has been decided that the buttons shall be used as follows:

left-hand button:	**select**
middle button:	**menu**
right-hand button:	**adjust**

The interpretation of which button should do what depends on the circumstances, but broadly speaking the **select** button is used to make new selections, while the **adjust** button is used to alter existing selections, or to add selections to existing ones.

Various parts of the Wimp enforce the interpretations given above for the mouse buttons. For example, icons may be programmed to respond in various ways to the **adjust** and **select** buttons, and the menu system uses the **menu** button for activation.

Keyboard input and text handling

An application running under the Wimp should perform all of its input using Wimp routines, rather than calling OS_ReadC or OS_Byte &81 directly. It is permissible for a program to scan the keyboard if necessary, using the OS_Bytes provided.

One window has what is termed the 'input focus'. For example, the main text window of an editor might be the current input window, and is highlighted by the Wimp to show this. Also, a caret (a vertical bar text cursor) is drawn in the input window at the current 'text insertion point'.

Two calls are provided – Wimp_SetCaretPosition and Wimp_GetCaretPosition – to set and read where the text insertion point appears within the window. An application reads key presses through the Wimp_Poll routine.

The caret can be positioned either inside a window's work area, or inside a writeable icon within a window. In the first case, all keys typed are passed on to the application, along with an indication of the caret position. It is up to the application to process the key presses appropriately.

In the second case, the Wimp will handle many possible key presses automatically (printable characters, and left and right arrow keys) by updating the contents of the icon and the caret position appropriately. The rest are passed on to the user. The Wimp will also respond to **select** and **adjust** button-presses over a writeable icon by positioning the caret inside that icon.

Pop-up menus

The Wimp provides a way in which the application can define multi-level menu selections. Once a menu has been activated, the Wimp takes care of all mouse movement over the menus, and when a selection is made, it will inform you through Wimp_Poll.

Because menus can have a complex hierarchical structure (as opposed to the simple single-level menus on some systems) a call Wimp_MenuDecode is provided to help translate the selection made into a textual form.

If the built-in menu handling is not suitable, the application can create its own menus by making selections from icons, and using the automatic icon highlighting and selection facilties that the Wimp provides.

Dialogue boxes

There is no direct way of setting up 'dialogue' boxes under the Wimp. However, because icons can be handled in very versatile ways, it is quite straightforward to set up windows which act as dialogue boxes. The Wimp can be made to deal with button clicks within the window, for example automatically highlighting icons.

Another feature of the Wimp which is useful in dialogue boxes is 'exclusive selection groups', where a highlighted icon is automatically de-highlighted if another icon from the same group is selected. This provides a 'radio button' facility, using the terminology of some Wimp systems.

Also, because writeable icons are available, it is a simple matter to input text from the user, again with the Wimp doing most of the work. If required, the application can restrict the movement of the mouse to within the dialogue box, by defining a mouse rectangle (using the pointer OS_Word described in the section **Mouse/pointer OS_Word call** in the chapter **THE VDU DRIVERS)** which encloses the box. This ensures that the user can perform no other task until he or she responds to the dialogue box. The application should always reset the mouse rectangle to the whole screen once the dialogue is over.

Dragging boxes

One of the recognisable features of most window systems is the ability to 'drag' items around the screen. The Arthur Wimp is no exception, and provides extensive facilities for dragging objects.

The call Wimp_DragBox initiates a dragging operation. The user supplies the initial position and size of the box to be dragged, and a 'parent' rectangle within which the dragging must be confined. Normally, the initial position of the box will be such that the mouse pointer is positioned along one of the box's edges. This is not mandatory though; the Wimp, while performing the dragging, ensures that the relative positions of the pointer and the box remain constant.

Drag boxes can be fixed size, where the whole of the box is moved along with the pointer, or variable sized, where the top-left of the box is fixed, and the bottom-right moves with the pointer. (The fixed and moveable corners can be reversed by specifying the box co-ordinates in the reverse order.)

Finally, there is an 'invisible' type of drag box. In this case, the mouse is simply constrained to the parent rectangle, and the initial box co-ordinates are ignored. It is up to the application to draw the object being dragged. This usually involves setting a 'dragging' flag in the main poll loop, and the use of Wimp_UpdateWindow. The application must also ensure that the dragged object is redrawn if a Redraw_Window_Request is issued.

In all cases, the application is notified when the drag operation ends (when the user releases all mouse buttons) by Wimp-Poll returning the reason code User_Drag_Box.

Tool windows and 'panes'

A pane is a window which is 'fixed' to another window, but has different properties from it. For example, consider a drawing program. You might have a scrollable, movable main window for the drawing area. This is called the tool window. On the left edge of this might be a fixed window which contains icons for the various drawing options. This left-hand window (the pane) always moves with the main window, but does not have scroll bars, or any other control areas.

Dealing with panes is really entirely up to the application program. However, there are one or two things to bear in mind when using them. If a tool window is closed, all of its panes must be closed too. Similarly, when a tool window is opened (an Open_Window_Request is received), the application must inspect the co-ordinates of the main window returned by the Wimp, and use them to open the pane in the appropriate position.

One bit in a window's definition is used to tell the Wimp that this is a pane. This is used by the Wimp in two circumstances:

- If the pane gets the input focus, the tool window is highlighted

- When toggling the tool window, the Wimp must treat panes as transparent.

Changing the pointer shape

You should not use the standard OS_Words and OS_Bytes to control the pointer shape under the Wimp. Instead, use the call, Wimp_SetPointerShape.

Pointer shape 1 is used by the Wimp as its default arrow pointer. Any program wishing to use a different shape must use shape 2, and program the pixels appropriately.

You should only change the pointer when it is within the work area of one of the application's windows. The Wimp_Poll routine returns two reason codes for detecting this: Pointer_Entering_Window and Pointer_Leaving_Window. Whenever the first code is received, the application can change the pointer to shape 2, and then, if required, later change it further, as long the pointer stays within the window. On receiving the second code, the application should reset the pointer to shape 1.

Template files

To facilitate the creation of windows, a 'template editor' has been written for the Wimp system. This allows you to use the mouse to design your own window layouts, and position icons as required. An extensive set of hierarchical menus provides a neat way of setting up all the relevant characteristics of the various windows and icons.

Once a window 'template' has been designed, it can be given an identifier (not necessarily the same as the window title) and saved (along with any other templates which have been set up and identified) in a template file. The Wimp provides a Wimp_LoadTemplate call, which makes it very simple for an application program, on start up, to load up a set of templates. The application can load a named template from the file, which can then be passed straight to Wimp_CreateWindow, or it can look for a wildcarded name, calling Wimp_LoadTemplate repeatedly for each match found.

There are two problems associated with the loading of window templates from a file. These concern the allocation of external resources:

- resolving references to 'indirected' icons
- resolving references to anti-aliased font handles.

In the first case, what happens is that the relevant indirected icon data is saved in the template file. When the template is loaded in, the application must provide a pointer to some free workspace where the Wimp can put the data, and redirect the relevant pointers to it. This pointer will be updated on exit from the call to Wimp_LoadTemplate. If there is not enough room, an error may be reported (the application must also provide a pointer to the end of the workspace).

The problem concerning font handles is more difficult to solve. The template file provides the binding from its internal font handles to the appropriate font names and sizes. In addition, the Wimp must also have some way of telling the application to which fonts handles it actually bound the font references to when the template was loaded. This is so the application can call Font_LoseFont as required when the window is deleted (or alternatively, when the application terminates).

To overcome this problem, the application must provide a pointer to a 256-byte array of font 'reference counts' when calling Wimp_LoadTemplate. Assuming each element of this array is zero on entry to Wimp_LoadTemplate, then if the template contains two references to font 3, and one to font 5, the Wimp will have called Font_FindFont three times altogether, receiving the answer 3 twice and the answer 5 once. On exit, element 3 of the array will contain 2, and element 5 will contain 1. Thus the application knows that when it has finished with the window (ie. if it deletes the window or terminates) it must call Font_LoseFont twice for font 3, and once for font 5. It is a question for the application writer whether it is sufficient to provide just one array of font reference counts, so that the fonts can be 'lost' only when all the windows are deleted (or the application terminates), or whether a separate array is needed for each window. Of course, considerable space optimisations could be made in the latter case if the array was scanned on exit from Wimp_LoadTemplate and converted to a more compact form.

If an application is confident that its templates do not contain references to anti-aliased fonts, then the array pointer can be null, in which case the Wimp reports an error if any font references are encountered.

WINDOW MANAGER SWIs

Wimp_Initialise &400C0

On entry: –

On exit: R0 = version number*100 (0 before v. 0.08)

This call initialises the system. It should be called just once, when the application starts up.

Wimp_CreateWindow &400C1

On entry: R1 = pointer to block

On exit: R0 = window handle

The block contains the following:

R1+0	initial visible area minimum x co-ordinate
R1+4	initial visible area minimum y co-ordinate
R1+8	initial visible area maximum x co-ordinate
R1+12	initial visible area maximum y co-ordinate
R1+16	scroll bar x offset
R1+20	scroll bar y offset
R1+24	handle to open window behind (–1 means top, –2 means bottom)
R1+28	flags/status information
R1+32	title foreground colour
R1+33	title background colour
R1+34	work area foreground colour
R1+35	work area background colour
R1+36	scroll bar outer colour
R1+37	scroll bar inner colour
R1+38	colour of title background when highlighted
R1+39	reserved – must be 0
R1+40	work area extent minimum x co-ordinate
R1+44	work area extent minimum y co-ordinate
R1+48	work area extent maximum x co-ordinate
R1+52	work area extent maximum y co-ordinate
R1+56	icon flags (type) for the title bar
R1+60	work area flags – for work area 'button type'
R1+64	sprite area control block pointer (0 for system area sprites)
R1+68	4 reserved bytes – must be &00000000
R1+72	title string
R1+84	number of icons in initial definition
R1+88	icon definitions (32 bytes each)

This call tells the window manager about the characteristics of a window. SWI Wimp_OpenWindow should subsequently be called to make it appear on the screen.

The work area button type (R1+60) determines how clicks over the work area are handled, in the same way as icon button type described below.

The window flags and status information are held in the bits of the four bytes R1+28 to R1+31 as follows:

Bit	Meaning when set
0	Window has title bar
1	Window is moveable
2	Window has vertical scroll bar
3	Window has horizontal scroll bar
4	Window can be redrawn entirely by the window manager (no user graphics)
5	Window is a pane (ie it is on top of a tool window)
6	Window is allowed to go outside the main area
7	Window has no 'back' or 'quit' boxes
8	'Scroll_Request' returned when scroll buttons clicked (auto-repeat)
9	'Scroll_Request' returned when scroll buttons clicked (debounced)
16	Window is open
17	Window is on top (ie not covered). This bit is set by the Wimp
18	Window has been toggled to full size. This bit is set by the Wimp

The handles of any icons defined in this call are numbered from zero upwards, in the same order that they appear in the block. The icon definitions are as supplied in SWI Wimp_CreateIcon below.

Note that this call may produce a `Bad work area extent` error, in which case the specified work area extent specified does not entirely contain the initial visible portion of the work area (governed by the scroll offsets and the work area co-ordinates).

Wimp_CreateIcon &400C2

On entry: R1 = pointer to block

On exit: R0 = icon handle (unique within that window)

The block contains the following:

R1+ 0	window handle
R1+ 4	minimum x co-ordinate of bounding box of icon
R1+ 8	minimum y co-ordinate of bounding box of icon
R1+12	maximum x co-ordinate of bounding box of icon
R1+16	maximum y co-ordinate of bounding box of icon
R1+20	flags
R1+24	icon data

This call provides all the information necessary to create an icon. Once it has been defined, you can change its flags by means of SWI Wimp_SetIconState, but you cannot alter the other data.

The bounding box of the icon is given relative to the window extent origin.

The flags are held in the bits of the four bytes R1+20 to R1+23 as follows:

Bit	**Meaning when set**
0	Icon contains text
1	Icon is a sprite
2	Icon has a border
3	Text is centred horizontally within the box
4	Text is centred vertically within the box
5	Icon has a filled background
6	Text is an anti-aliased font
7	Icon requires application's help to be redrawn
8	Icon data is 'indirected' (see below)
9	Text is right-justified within the box

10	If selected with right-hand button don't cancel others in the same exclusive selection group
11	Reserved (must be 0)
12 – 15	Button type, way in which icon responds to mouse clicks
16 – 20	Exclusive selection group (ESG)
21	Icon is selected by the user and is inverted
22	Icon cannot be selected by the mouse pointer, it is shaded
23	Icon has been deleted
24 – 27	Foreground colour of icon (if bit 6 is cleared)
28 – 31	Background colour of icon (if bit 6 is cleared)
24 – 31	Font number (if bit 6 is set)

The icon data consists of 12 bytes which contain:

– if text, then up to 12 bytes of text (terminated by <cr>)

– if sprite, then the name of the sprite (12 bytes)

– if icon data is 'indirected', then the following 3 words:

– pointer to buffer to contain the text

– pointer to validation string (–1 if none – use this for now)

– length of buffer (bytes)

The button types (bits 12 – 15) are as follows:

Value	**Type**
0	Ignore mouse clicks
1	Notify application whenever pointer is over this icon
2	Click notifies application (auto-repeat)
3	Click notifies application (debounced)
4	Click selects, release notifies application (or deselect if have moved away from icon)
5	Click selects, double-click notifies application

6	As (3), but can also drag (returns button state * 16)
7	As (4), but can also drag (returns button state * 16)
8	As (5), but can also drag (returns button state * 16)
9	Select when mouse pointer is over icon, notify if clicked
10	Click returns button state*256 Drag returns button state*16 Double click returns button state*1
11 – 14	Reserved
15	Writeable icon (mouse clicks cause the caret to be positioned inside the icon)

Once an icon has been defined, you can change its flags by means of the SWI Wimp_SetIconState instruction, but the other data (the bounding box and the text) cannot be altered. If the text is specified as 'writeable' then it can be altered, but the application must ensure that the icon is redrawn (for example, by using a call to SWI Wimp_SetIconState which leaves the flags unchanged).

If an icon has a button type of 15, then it is treated as being special by the window manager. Any mouse clicks on it cause the caret to be automatically positioned inside the icon, whereupon any key presses cause the text in the icon to be automatically updated. For further details, see the sections on SWI Wimp_SetCaretPosition, SWI Wimp_GetCaretPosition, and SWI Wimp_Poll (Key_Pressed).

The exclusive selection group number (ESG) groups sets of icons such that if any one of them is selected, then the others are automatically deselected. Another bit in the icon flags allows the use of the right-hand button (adjust) to select more than one item in an ESG. If the ESG number of an icon is set to zero, then it is in a group on its own, so clicking again on such an icon will deselect it.

If it is not possible for the Wimp automatically to select or deselect an icon, because it requires user-graphics for example, then you can use type 10. This allows the application to trap single, double clicks, and dragging, and act on them accordingly. Note that ESG handling is only performed by the Wimp if selection is performed automatically; it won't turn off other icons in the same ESG if you use Wimp_SetIconState to highlight an icon. This is to give you total control over highlighting if you are doing it 'manually'. Wimp_WhichIcon helps here: it will give you a list of icons in a given ESG, and the handles which are returned can be used to perform highlighting as required.

Wimp_DeleteWindow &400C3

On entry:	R1 = pointer to block
On exit:	–

The block contains the following:

R1+ 0	window handle

This call removes the definition of the specified window from the data structure, along with the definitions of all the icons within it. The memory used is reallocated.

Wimp_DeleteIcon &400C4

On entry:	R1 = pointer to block
On exit:	–

The block contains the following:

R1+ 0	window handle
R1+ 4	icon handle

This call removes the definition of the specified icon from the data structure. The icon is marked as deleted if it is not the last one in the list, so that the handles of the other icons are not altered. If the icon is the last one in the window's list, the memory is reclaimed.

Note that this call doesn't affect the screen. To cause the icon to disappear, you must call Wimp_ForceRedraw with the bounding box of the icon, or the area occupied by a group of icons if several have been deleted.

Wimp_OpenWindow &400C5

On entry: R1 = pointer to block

On exit: –

The block contains the following:

R1+ 0	window handle
R1+ 4	minimum x of work area
R1+ 8	minimum y of work area
R1+12	maximum x of work area
R1+16	maximum y of work area
R1+20	x offset of top left of visible work area from graphics origin
R1+24	y offset of top left of visible wrok area from graphics origin
R1+28	handle of window to go behind (–1 = top, –2 = bottom)

This call causes the appropriate window to appear on the screen in the specified position, at the specified size, and with the scroll bars in the appropriate position. The last parameter controls the position of the window in the window stack.

This call provides a means of moving windows about the screen. Any necessary redrawing is returned to the application program via SWI Wimp_Poll, unless the window manager can do it itself, in which case it is done immediately.

When the window manager returns an Open_Window_Request from SWI Wimp_Poll, the last parameter is set up according to the nature of the action which is causing the window to move. For example: 'back window', 'move window', 'resize window', 'scroll up', etc.

Wimp_CloseWindow &400C6

On entry: R1 = pointer to block

On exit: –

The block contains the following:

R1+ 0	window handle

This call removes the specified window from the active list. It may cause SWI Wimp_Poll subsequently to return Redraw_Window_Requests as appropriate to update the screen.

Wimp_Poll &400C7

On entry: R0 = mask
R1 = pointer to block (used for return data)

On exit: R0 = reason code
R1 = pointer to block (data depends on reason code)

This call checks to see whether certain events have occurred.

If the mask is zero on entry, all reason codes are checked for. If non-zero, then the events, corresponding to the bits that are set, are not checked for. They cannot be returned by the window manager. These events are as follows:

Bit	**Meaning when set**
0	Disallow Null_Reason_Code
1	Disallow Redraw_Window_Request
2	Disallow Open_Window_Request †
3	Disallow Close_Window_Request †
4	Disallow Pointer_Leaving_Window
5	Disallow Pointer_Entering_Window
6	Disallow Mouse_Button_Change
7	Disallow User_Drag_Box †
8	Disallow Key_Pressed
9	Disallow Menu_Select †
10	Disallow Scroll_Request †

† These codes cannot be masked out.

Possible reason codes are checked for in numerical order, so that, for example, a Redraw_Window_Request takes precedence over a Mouse_Button_Change (unless it is masked out).

The following reason codes which may be returned:

Code	Reason
0	Null_Reason_Code
1	Redraw_Window_Request
2	Open_Window_Request
3	Close_Window_Request
4	Pointer_Leaving_Window
5	Pointer_Entering_Window
6	Mouse_Button_Change
7	User_Drag_Box
8	Key_Pressed
9	Menu_Select
10	Scroll_request

According to the reason code returned, the buffer pointed to by R1 contains the relevant data, allowing the application to respond appropriately. The format of the data in the block for each of the possible reason codes is as follows:

Redraw_Window_Request

The block contains the following:

R1+ 0	window handle

In general, when this code is returned, not all the relevant window needs to be redrawn, but just the portion of the window which is not up to date. This portion consists of a series of rectangles.

On receipt of the Redraw_Window_Request, the application must first ask the window manager to redraw the relevant parts of the window outline. Then it must redraw the work area of the window for each of the rectangles in the 'invalid list'.

Two SWIs are provided to help with this, SWI Wimp_RedrawWindow and SWI Wimp_GetRectangle:

- Wimp_RedrawWindow causes the Wimp to redraw any sections of the window outline which intersect the invalid list, and also to clear any relevant parts of the window's work area to the background colour specified in the window's definition. It then works out which portion of the window's work area intersects the invalid list, and exits via Wimp_GetRectangle.

- Wimp_GetRectangle returns the next rectangle in the list which specifies the invalid portion of the window's work area. It sets the graphic window to that rectangle, and returns the relevant co-ordinates in the user's buffer. This allows the application to redraw only those parts of its work area which it knows intersect with the given rectangle.

The code needed in response to a Redraw_Window_Request is of the following type:

```
<call Wimp_RedrawWindow> – returns flag <more_rectangles>
WHILE
<more_rectangles>
<redraw the work area>
<call Wimp_GetRectangle> – returns flag <more_rectangle>
ENDWHILE
```

Open_Window_Request

The block contains the following:

R1+ 0	window handle
R1+ 4	new minimum x window co-ordinate
R1+ 8	new minimum y window co-ordinate
R1+12	new maximum x window co-ordinate
R1+16	new maximum y window co-ordinate
R1+20	new x scroll bar position
R1+24	new y scroll bar position
R1+28	handle of window to put this one behind

This reason code is returned as a result of the size change or title box of a window being selected or either of the scroll bars being dragged to a new position. The dragging process is performed by the window manager itself. When it has finished, the 'Open Window' code is returned to the application. SWI Wimp_OpenWindow should then be called with the desired new attributes. The next call of SWI Wimp_Poll returns the Redraw_Window_Request code to instruct the application to call SWI Wimp_RedrawWindow and redraw the work area.

Note that the data returned with this reason code is in the correct format for SWI Wimp_OpenWindow, so the application can use the same parameter block.

Close_Window_Request

The block contains the following:

R1+ 0	window handle

This reason code is returned when you click (the mouse) on the 'quit' box of a window.

The application should normally call SWI Wimp_CloseWindow. If, however, you do not want the window to close, the application could do something else instead. For example, it could open an error box explaining why the window should not be closed, or ask the user for confirmation. Alternatively, the closing of one window might cause another one to be closed as well, in which case two calls of SWI Wimp_CloseWindow would be needed.

Pointer_Leaving_Window

The block contains the following:

R1+ 0	window handle

This reason code is returned when the pointer has moved away from a window. This is useful for handling pop-up menus, which can be made to disappear as soon as the mouse pointer leaves them.

Pointer_Entering_Window

The block contains the following:

R1+ 0	window handle

This reason code is returned when the pointer has moved onto a window. This is useful for handling windows which are activated without the need for a button being pressed.

Mouse_Button_Click

The block contains the following on exit:

R1+0	mouse x-co-ordinate
R1+4	mouse y-co-ordinate
R1+8	new state of buttons
R1+12	window handle (or –1 if none)
R1+16	icon handle (or –1 if none)
R1+20	old state of buttons

This reason code is returned when the state of the mouse buttons has been altered. The new state of the buttons is as follows:

Bit	**Meaning**
0	right-hand button pressed (adjust)
1	middle button pressed (menu)
2	left-hand button pressed (select)
4	drag initiated with the adjust
6	drag initiated with the select
8	single click with adjust (if icon/work area button type = 10)
10	single click with select (if icon/work area button type = 10)

The window handle indicates which window the mouse pointer was over when the button change took place, and similarly the icon handle indicates which icon it was over. Note that the window manager only returns to the application with this reason code if the conditions demanded by the 'button type' of the icon have been met; for example if a double-click has occurred on an icon of type 5 (see SWI Wimp_CreateIcon for a full list of button types).

Operations such as highlighting an icon when it is selected and the cancellation of the other selections in the same ESG are all done automatically by the window manager, unless the button type is 10, in which case the click type is always returned to the application.

Note that any system operations performed by clicking the mouse over a window's scroll bar, for instance, are also transparent to the application and do not cause this reason code to be returned.

User_Drag_Box

The block contains the following:

R1+0	drag box minimum x co-ordinate
R1+4	drag box minimum y co-ordinate
R1+8	drag box maximum x co-ordinate
R1+12	drag box maximum y co-ordinate

This reason code is returned when you have finished doing a User_Drag operation. This operation starts when the application issues a SWI Drag_Box with a drag type of 5, 6 or 7. It finishes when you release all the mouse buttons (at which point this reason code is returned).

During a drag operation (particularly with drag type 7), the application program may wish to keep track of the mouse pointer. To do this, it should use SWI Wimp_GetPointerInfo to read the mouse position, returning to the central polling routine between each reading of the mouse position. When the SWI Wimp_Poll code 7 is received, the application should cancel its internal 'dragging state' flag.

Key_Pressed

The block contains the following:

R1+ 0	window handle where caret is
R1+ 4	icon handle
R1+ 8	x-offset of caret (within window)
R1+12	y-offset of caret
R1+16	caret height (OS co-ordinates)
R1+20	index of caret inside string
R1+24	character code of key pressed

This reason code is returned to the application when a character is pressed which is relevant to one of its windows, ie the 'input focus'.

The first 5 words in the block are exactly as they would be for the SWI Wimp_GetCaretPosition call. If the icon handle is –1, then the key press is always returned to the application (if it owns the relevant window).

If the caret is inside a writeable icon the Wimp will attempt to process the key itself, and will only return to the application if it does not know what to do with it. This approach allows the application to deal with other control keys: for example, carriage return could cause the caret to move to the start of the next line. The Wimp understands the following keys:

Normal characters	(&20 – &7E, &80 – &FF). These are printed as usual
Delete	delete character to left of caret
Copy	delete character to right of caret
←	move left one character
→	move right one character
Shift Copy	delete word (forwards)
Shift ←	move left one word (returns code if at left of line)
Shift →	move right one word (returns code if at right of line)
Ctrl Copy	delete forwards to end of line
Ctrl ←	move to left end of line
Ctrl →	move to right end of line

In order to avoid clashes between top bit set characters and function key codes, the Wimp employs a facility in the OS which allows the function keys to return two-character pairs, with the first character being a zero and the second character the key code. In this way it is possible to distinguish between a function key and the top-bit-set characters, obtained by pressing Ctrl Shift Alt followed by a key. To make things easier for applications, the Wimp traps the two-character combinations, and modifies the code returned by adding &100, so the codes returned for the 'special' keys are as follows:

Copy	&18B	&19B	&1AB	&1BB
←	&18C	&19C	&1AC	&1BC
→	&18D	&19D	&1AD	&1BD
↓	&18E	&19E	&1AE	&1BE
↑	&18F	&19F	&1AF	&1BF
f0 – f9	&180 – &189	&190 – &199	&1A0 – &1A9	&1B0 – &1B9
f10 – f12	&1CA – &1CC	&1DA – &1DC	&1EA – &1EC	&1FA – &1FC
Tab	&18A	&19A	&1AA	&1BA
Insert	&1CD	&1DD	&1ED	&1FD
Print	&180	&190	&1A0	&1B0 (same as f0)
Page Down	&19E	&18E	&1BE	&1AE
Page Up	&19F	&18F	&1BF	&1AF

To tell the operating system to return the appropriate codes, Wimp_Initialise programs the soft key expansion and cursor key codes by performing the following OS_Byte calls:

```
*FX 4,2 *FX 221,2 up to *FX 228,2
```

Applications running under the Wimp are not allowed to change any of these settings. (So soft key expansions are not allowed. This isn't too much of a disadvantage, as the application can still access the key's expansion string using the key$n variables.)

Note

Versions of the Wimp before 0.18 do not use this method. Instead, they use the *FX4,1 option to control the codes returned by the cursor keys, and so return the following values:

Copy	&87	(same code if Ctrl or Shift are pressed, too)
←	&88	
→	&89	
↓	&8A	
↑	&8B	

The soft key settings and the code returned by the Tab key are not defined at all.

The result of this is that if a program needs to access the function/cursor/tab keys, then it must either specifically complain if the Wimp version number is less than 0.18, or it must perform a series of 'bodges' to maintain compatibility with all versions. A suitable program listing is given at the end of this document.

Menu_Select

The block contains the following:

R1+0	item number in first menu which was selected (starting at 0)
R1+4	item number in second menu which was selected
	....
	(terminated by –1)

To 'pop-up' a set of menus, the application should call SWI Wimp_CreateMenu and then return to its normal polling routine, having set its own suitable flag so that it knows which set of menus the Menu_Select code from SWI Wimp_Poll relates to (when it eventually arrives).

The values in the block indicate which item in each sub-menu was selected by the user (the first item in each menu is item 0), with a –1 entry to terminate the list. Note that it is possible for the user to specify an ambiguous command, by clicking on an item which itself has sub-menus. In this case, the command may be meaningless, or it may be possible for the application to use the previously-selected values for the items which were not specified explicitly.

Scroll_Request

The block contains the following:

R1+0	window handle
R1+4	work area minimum x
R1+8	work area mininum y
R1+12	work area maximum x
R1+16	work area maximum y
R1+20	scroll bar x position
R1+24	scroll bar y position
R1+28	handle of window to open behind
R1+32	scroll direction x (see below)
R1+36	scroll direction y (see below)

This reason code is returned if the user clicks on one of the scroll buttons of a window which has one of the 'Scroll_Request returned' bits set. It indicates which direction to scroll in, instead of simply returning with an Open_Window_Request at the new scroll offset.

It returns the old scroll bar positions, as the amount to scroll is up to the application. The scroll directions have the following meanings:

Value	Meaning	
–2	Page left/up	(v. 0.18 and above)
–1	Left/up	
0	No change	
+1	Right/down	
+2	Page right/down	(v. 0.18 and above)

Page scrolls are returned when the user clicks in the scroll bar outside of the scroll 'sausage'.

When this reason code is received, the application should decide how much to scroll by (by scaling the scroll directions appropriately), update the scroll offsets by these amounts, and issue an Wimp_OpenWindow command.

Wimp_RedrawWindow &400C8

On entry: R1 = pointer to block

On exit: R0 = flag

The block contains the following on entry:

R1+ 0	window handle

The block contains the following on exit:

R1+ 0	window handle
R1+ 4	work area minimum x co-ordinate
R1+ 8	work area minimum y co-ordinate
R1+12	work area maximum x co-ordinate
R1+16	work area maximum y co-ordinate
R1+20	x scroll position
R1+24	y scroll position
R1+28	current graphics window minimum x co-ordinate
R1+32	current graphics window minimum y co-ordinate
R1+36	current graphics window maximum x co-ordinate
R1+40	current graphics window maximum y co-ordinate

This call redraws the window, whose handle is passed in R1+0 – R1+3, in its current position, where any visible parts of it intersect with the invalid rectangle list. The border area is drawn automatically. The routine exits via Wimp_GetRectangle so that the first rectangle of the work area to be drawn (if any) is returned. On exit, R0 contains a flag as follows:

Flag	Meaning
True	First rectangle to be drawn has been set up
False	There are no rectangles to be drawn

The application program should call this routine after a SWI Wimp_Poll call has returned a code requesting a redraw. The relevant graphics clip window is set up when the routine is called and on each subsequent call to SWI Wimp_GetRectangle. Note that although the application could just redraw its entire work area for each of the rectangles returned, it is much more efficient for it to take note of the graphics clip window co-ordinates when working out what it must redraw.

Note that any redrawing must use VDU 5 mode to print characters, since any graphic operations must be clipped to the current graphics window (set up by SWI Wimp_GetRectangle). The window manager performs a VDU 5 when SWI Wimp_RedrawWindow is called automatically.

Wimp_UpdateWindow &400C9

On entry: R1 = pointer to block

On exit: R0 = flag

The block contains the following on entry:

R1+ 0	window handle
R1+ 4	work area extent minimum x co-ordinate
R1+ 8	work area extent minimum y co-ordinate
R1+12	work area extent maximum x co-ordinate
R1+16	work area extent maximum y co-ordinate

These co-ordinates give the area of the work area which is to be marked as 'invalid' and therefore must be updated.

The block contains the following on exit:

R1+0	window handle
R1+4	work area minimum x co-ordinate
R1+8	work area minimum y co-ordinate
R1+12	work area maximum x co-ordinate
R1+16	work area maximum y co-ordinate
R1+20	x scroll position
R1+24	y scroll position
R1+28	current graphics window minimum x co-ordinate
R1+32	current graphics window minimum y co-ordinate
R1+36	current graphics window maximum x co-ordinate
R1+40	current graphics window maximum y co-ordinate

This call is similar to Wimp_RedrawWindow. However, it returns all visible rectangles of the window, regardless of whether or not they are invalid.

When calling this routine, the application can assume that the data inside the window is intact and can be modified.

As with SWI Wimp_RedrawWindow, the application should perform its updating inside a WHILE loop, calling Wimp_GetRectangle to get subsequent rectangles. Also, as with SWI Wimp_RedrawWindow, the application must use VDU 5 mode when printing characters (or use the anti-aliased font system).

Wimp_GetRectangle &400CA

On entry: R1 = pointer to block

On exit: R0 = flag

The block contains the following:

R1+0	window handle
R1+4	work area minimum x co-ordinate
R1+8	work area minimum y co-ordinate
R1+12	work area maximum x co-ordinate
R1+16	work area maximum y co-ordinate
R1+20	x scroll position

R1+24	y scroll position
R1+28	current graphics window minimum x co-ordinate
R1+32	current graphics window minimum y co-ordinate
R1+36	current graphics window maximum x co-ordinate
R1+40	current graphics window maximum y co-ordinate

This call returns the details of the next rectangle of the work area to be drawn (if any). The details returned are in exactly the same format as those returned by Wimp_RedrawWindow and Wimp_UpdateWindow above.

Note that the window handle will be faulted by the window manager if it differs from the one last used when Wimp_RedrawWindow or Wimp_UpdateWindow was called. This means that an application must redraw the whole of a window before performing any other operations.

Wimp_GetWindowState &400CB

On entry: R1 = pointer to block

On exit: –

The block contains the following on entry:

R1+0	window handle

The block contains the following on exit:

R1+0	window handle
R1+4	work area minimum x co-ordinate
R1+8	work area minimum y co-ordinate
R1+12	work area maximum x co-ordinate
R1+16	work area maximum y co-ordinate
R1+20	x scroll position
R1+24	y scroll position
R1+28	handle of window in front of this one (–1 if on top)
R1+32	flags/status information

This call is used to find out the current values of transient variables associated with the window whose handle is given. Usually, the co-ordinates of the window can be ascertained without using this call, since SWI Wimp_RedrawWindow and SWI Wimp_UpdateWindow return the window co-ordinates anyway. However, this call is useful for finding out if a window is on top, by looking at the appropriate bit in the flags/status word at R1+32.

The window flags and status information held in the bits of the four words R1+32 – R1+35 are as follows:

Bit	Meaning when set
0	Window has title bar
1	Window is moveable
2	Window has vertical scroll bar
3	Window has horizontal scroll bar
4	Window can be redrawn entirely by the window manager (no user graphics)
5	Window is allowed to go outside the screen
16	Window is open
17	Window is on top (ie not covered)

Wimp_GetWindowInfo &400CC

On entry: R1 = pointer to block

On exit: –

The block contains the following on entry:

R1+0	window handle

The block contains the following on exit:

R1+0	window handle
R1+4	work area minimum x co-ordinate
R1+8	work area minimum y co-ordinate
R1+12	work area maximum x co-ordinate
R1+16	work area maximum y co-ordinate
R1+20	x scroll position
R1+24	y scroll position
R1+28	handle of window in front of this one (–1 if on top)
R1+32	flags/status information
R1+36	title foreground colour
R1+37	title background colour
R1+38	work area foreground colour
R1+39	work area background colour
R1+40	scroll bar outer colour
R1+41	scroll bar inner colour
R1+42	colour of title background (highlight colour)
R1+43	reserved
R1+44	work area extent minimum x co-ordinate
R1+48	work area extent minimum y co-ordinate
R1+52	work area extent maximum x co-ordinate
R1+56	work area extent maximum y co-ordinate
R1+60	icon flags for the title bar
R1+64	work area flags
R1+68	sprite area control block pointer
R1+72	reserved
R1+76	title string
R1+88	number of icons in initial definition
R1+92	icon definitions (32 bytes each)

You can use this to find out information about the window whose handle is given.

Wimp_SetIconState &400CD

On entry: R1 = pointer to block

On exit: –

The block contains the following:

R1+0	window handle
R1+4	icon handle
R1+8	word to EOR with old icon state
R1+12	word to BIC with old icon state (mask)

This call allows the icon status to be set as follows:

<new state> = (<old state> BIC <mask word>) EOR <EOR word>

The mask word allows the application to change certain parts of the icon status without affecting others, simplifying the process of changing colours, for example. This call also results in the updated state of the icon being reflected on the screen, if appropriate.

Note that it is not possible to change the co-ordinates or textual data associated with an icon with this call. To do this, the icon must first be deleted and then recreated.

State

Wimp_GetIcon~~Info~~ &400CE

On entry: R1 = pointer to block

On exit: –

On entry the block contains the following:

R1+ 0	window handle
R1+ 4	icon handle

On exit the block contains the following:

R1+0	window handle
R1+4	icon handle
R1+8	minimum x co-ordinate of bounding box of icon (relative to extent origin)
R1+12	minimum y co-ordinate of bounding box of icon (relative to extent origin)
R1+16	maximum x co-ordinate of bounding box of icon (relative to extent origin)
R1+20	maximum y co-ordinate of bounding box of icon (relative to extent origin)
R1+24	flags
R1+28	icon data (sprite name or text)

This call may be used to find out information about an icon.

The flags are held in the bits of the four words R1+16 – R1+19 as follows:

Bit	**Meaning when set**
0	Icon contains text
1	Icon is a sprite
2	Icon has a border
3	Text is centred horizontally within the box
4	Text is centred vertically within the box
5	Icon has a filled background
6	Text is an anti-aliased font
7	Icon requires application's help to be redrawn
8	Icon contains a 'writeable' text field
9	Text is right-justified within the box
10	If selected with right-hand button don't cancel others in same exclusive selection group
11	Reserved
12 – 15	Button type, ie way in which icon responds to mouse clicks
16 – 20	Exclusive selection group (ESG)
21	Icon is selected by the user
22	Icon cannot be selected by the mouse pointer
23	Icon is deleted

24 – 27	Foreground colour of icon (or font number if bit 6 is set)
28 – 31	Background colour of icon (or font number if bit 6 is set)

The icon data consists of 12 bytes which contain text if the icon contains text, or a sprite name if the icon is a sprite.

If you want to search for an icon with particular flag settings, for example, to find out which icon in a group has been selected), you should use SWI Wimp_WhichIcon.

Wimp_GetPointerInfo &400CF

On entry: R1 = pointer to block

On exit: –

The block contains the following:

R1+ 0	mouse x co-ordinate
R1+ 4	mouse y-co-ordinate
R1+ 8	mouse button state
R1+12	window handle (–1 if not over a window)
R1+16	icon handle (–1 if not over an icon)

This call returns information about the state of the pointer and the mouse buttons. It enables the application to find out where the mouse pointer is independently of the buttons being pressed or released, for example, for tracking purposes.

The status of the buttons as returned in R1+8 – R1+ 11 is indicated by the bit pattern as follows:

Bit	**Meaning if set**
0	Right-hand button pressed (adjust)
1	Middle button pressed (menu)
2	Left-hand button pressed (select)

Wimp_DragBox &400D0

On entry: R1 <= 0 to cancel draw_box, otherwise:
R1 = pointer to block

On exit: –

On entry the block contains the following:

R1+ 0	window handle
R1+ 4	drag type
R1+ 8	minimum x co-ordinate of initial position of drag box
R1+12	minimum y co-ordinate of initial position of drag box
R1+16	maximum x co-ordinate of initial position of drag box
R1+20	maximum y co-ordinate of initial position of drag box
R1+24	minimum x co-ordinate of parent box (for codes 5 – 7)
R1+28	minimum y co-ordinate of parent box (for codes 5 – 7)
R1+32	maximum x co-ordinate of parent box (for codes 5 – 7)
R1+34 36	maximum y co-ordinate of parent box (for codes 5 – 7)

This call causes the definded box to move as the pointer moves until all the mouse buttons are released. The action depends on the drag type as follows:

Drag type	Meaning
1	Change position of window
2	Change size of window
3	Drag horizontal scroll bar
4	Drag vertical scroll bar
5	User drag box – fixed size box
6	User drag box – 'rubber' box
7	User drag box – invisible box

Types 1 – 4

These are the 'system' types since they relate to picking up a window, changing its size and scrolling it respectively. In these cases, the bounding box for pointer movement is worked out by the window manager. In the case of type 2, the bounding box is determined by the maximum and minimum sizes of the window as defined when it was created. These are calculated automatically by the Wimp.

When all the buttons are released, an Open_Window_Request is returned.

Types 5 – 7

These are 'user' types, where the application decides what the significance of the dragging will be. In these cases, you supply the co-ordinates of the parent box. In the case of type 7, where there is no inner box to be dragged, the initial drag box position is ignored and the mouse co-ordinates are used instead.

When all the buttons are released, a User_DragBox is returned.

Wimp_ForceRedraw &400D1

On entry: R0 = window handle (–1 = whole screen)
R1 = minimum x co-ordinate of area to redraw
R2 = minimum y co-ordinate of area to redraw
R3 = maximum x co-ordinate of area to redraw
R4 = maximum y co-ordinate of area to redraw

This call forces an area of a window or the screen to be marked as invalid.

If R0 is –1 on entry, then that area of the screen specified in absolute co-ordinates is marked invalid. This causes it to be redrawn later.

If R0 is not –1, then it indicates a window handle, and the area of the window specified relative to the window's work area origin is marked invalid (if it is visible).

This call is useful either for reconstructing the screen if for some reason it has been corrupted, or for reinstating a particular area after, for example, an error box has been drawn over the top of it. Other uses include redrawing the screen after redefining one or more of the soft characters, which could affect any part of the screen.

Two strategies are possible when the application is required to change the contents of a window. These are:

– Call this routine, which causes the specified area to be redrawn later

– Call Wimp_UpdateWindow, followed by the necessary graphic operations (and calls to Wimp_GetRectangle).

The latter method is generally quicker, but involves more code.

Wimp_SetCaretPosition &400D2

On entry: R0 = window handle (–1 if caret is to be turned off)
R1 = icon handle (–1 if none)
R2 = x-offset of caret (relative to window origin)
R3 = y-offset of caret (relative to window origin)
R4 = height of caret (if –1, then R2, R3, R4 are calculated from R0,R1,R5)
R5 = index into string (if –1, then R4, R5 are calculated from R0,R1,R2,R3)

This call sets up the new data for the caret position. It also removes the caret from its old position and redraws it in the new position.

The exact meaning of R4 is as follows:

Bits	Meaning
0 – 23	Height of caret in OS co-ordinates
24	If set, a VDU 5-type caret is used, else anti-aliased caret
25	If set, the caret is invisible; the application must draw it

By setting R4 (height) to –1, it is possible to make the window manager calculate the x,y co-ordinates of the caret, and its height (R2, R3, R4) from the data in R0, R1 and R5. This is only possible if R1 contains an icon handle.

Similarly, by setting R5 (the index) to –1, it is possible to make the window manager calculate the index into the string, and the caret height (R4, R5) from R0 – R3.

In each case, the height of the caret is determined from the bounding box of the font used in the icon (if the icon contains normal text, the caret height is determined appropriately).

Hence, to set up a 'writeable icon', you should create it with the following flag settings:

- choose either normal text or an anti-aliased font
- choose between left-justified, centred or right-justified text
- set the 'button type' to 15, ie a writeable icon
- set the 'indirected' bit if you want to supply the text buffer address – if not, the text is limited to 12 chars (including terminator)
- the icon does not have to have a filled background, but it is treated as filled when the window manager updates the text inside it.

Wimp_GetCaretPosition &400D3

On entry:	R1 = pointer to block
On exit:	–

The block contains the following:

R1+ 0	window handle (–1 if caret is turned off)
R1+ 4	icon handle (–1 if none)
R1+ 8	x-offset of caret (relative to window origin)
R1+12	y-offset of caret (relative to window origin)
R1+16	height of caret
R1+20	index into string

The parameters returned by this call correspond to those supplied by SWI Wimp_SetCaretPosition, and are also the same as the first five parameters returned by the Key_Pressed return from SWI Wimp_Poll.

Wimp_CreateMenu &400D4

On entry:	R1 = –1 means close all menus, or
	R1 = pointer to block
	R2 = x co-ordinate of top-left of top menu
	R3 = y co-ordinate of top-left of top menu
On exit:	–

The block contains the following:

R1+ 0	menu title (if null, then menu is untitled)
R1+12	menu title foreground colour
R1+13	menu title background colour
R1+14	menu work area foreground colour
R1+15	menu work area background colour
R1+16	width of following menu items
R1+20	height of following menu items
R1+24	vertical gap between items (and at top and bottom of menu)
R1+28	menu items (each 24 bytes):

0 – 3	menu flags:	
	&01	Wimp will display a 'tick' to the left of the item
	&02	Dotted line following (separates sections)
	&04	Item is 'writeable' for text entry (v.0.14 and above)
	&80	This is the last item in this menu
4 – 7	sub-menu pointer or sub window handle (–1 if none)	
8 – 11	menu icon flags – as for a normal icon	
12 – 23	menu icon data (12 bytes) – as for a normal icon	

The most important flags are the 'shaded' bit, which can be used to make an item non-selectable, and the 'tick' bit, which should be used to show which, if any, are the default values. For example, for a font selection, the font and size previously set up for the appropriate piece of text.

Having made this call, the application should return to its normal polling routine. The window manager creates the top menu described by the structure, and, while the application is calling SWI Wimp_Poll, maintains the various operations that can occur in connection with the menu structure. The sub-menu pointer for a menu item, if not –1, points to a similar data structure describing a sub-menu which is automatically popped-up by the window manager if the user positions the mouse pointer over the appropriate icon. Menu items with a non-null sub-menu pointer have a right arrow displayed to the right of them, which activates the sub-menu.

If the pointer is in fact a window handle, this window is opened (as if it were a menu) when the mouse pointer moves over the arrow. It is restored to its original state afterwards.

If the user moves the mouse pointer onto a menu item which is marked as 'writeable', then the caret will be automatically positioned inside the appropriate item, whereupon the user can enter data as required. If ↵ is pressed when the caret is inside such an icon, it will be treated by the Wimp as though a mouse button had been pressed, ie the item is selected.

The window manager takes care of the menus until the user makes a click with any of the mouse buttons. If the click was outside the menus, then the window manager closes all the menus and treats the mouse click as if they had not been there. If the mouse is clicked inside the menus, then a Menu_Select reason code is returned from SWI Wimp_Poll, along with a list of selections.

If the application creates more than one type of menu, it must set up a flag when Wimp_CreateMenu is called in order to determine what to do next. It must also scan down its data structure to determine which sub-menus the numbers relate to.

It is recommended that applications provide a 'shorthand' for defining menus, which is translated into the full form required by the window manager when needed.

Wimp_DecodeMenu &400D5

On entry: R1 = pointer to menu data structure
R2 = pointer to a list of menu selections
R3 = pointer to a buffer to contain the answer

On exit: R3 = pointer to a string, being menu items separated by '.'

On receipt of a Wimp_Poll: Menu_Select reason code, there are at least two main approaches:

– use a series of nested CASE statements to decode the result

– use SWI Wimp_DecodeMenu to provide a string equivalent, and decode that.

It is also possible to use a combination of these methods, for instance if one of the possible sub-menus from the main menu is to select a font with a hierarchical name.

Wimp_WhichIcon &400D6

On entry: R0 = window handle
R1 = pointer to block to contain the list of icon handles
R2 = bit mask (bit set ==> consider this bit)
R3 = desired bit settings

On exit: R1 = pointer to a list of icon handles (1 word each, terminated by –1)

This call can be used to detect which of a group of icons has been selected, or other things depending on how the mask is set. For example:

```
SYS "Wimp_WhichIcon",<handle>,buffer%,&00200000,&00200000
```

On exit the list of selected icon handles will be in the buffer.

To see which is the first icon in a particular ESG to be selected, perform a SWI Wimp_WhichIcon. For example, if the ESG number is 1:

```
SYS "Wimp_WhichIcon",<handle>,buffer%,&003F0000,&00210000
```

!buffer% now contains the handle of the required icon, or –1 if none is selected.

Wimp_SetExtent &400D7

On entry: R0 = window handle
R1 = pointer to block

On exit: –

On entry, the block contains:

R1+ 0	new work area extent minimum x
R1+ 4	new work area extent minimum y
R1+ 8	new work area extent maximum x
R1+ 12	new work area extent maximum y

This call sets the work area extent of the specified window, and usually causes the window's scroll bars to be redrawn (to reflect the new total size of window). The work area extent may not be changed so that any part of the visible portion of the work area lies outside the extent, so this call cannot change the current size of a window, or cause it to scroll.

Wimp_SetPointerShape &400D8

On entry: R0 = shape number (0 for pointer off)
R1 = pointer to shape data (–1 for no change)
R2 = width in pixels (must be multiple of 4)
R3 = height in pixels
R4 = active point x offset from top-left in pixels
R5 = active point y offset from top-left in pixels

On exit: –

The shape data is a series of bytes giving the pixel colours for the shape. Each row of the shape is given as a whole number of bytes (eg 3 bytes for a 12-pixel wide shape). Bytes are given in left to right order. The least significant two bits of each byte give the colour of the left-most pixel in that group of four.

This convention should be used when programming the pointer shape under the Wimp:

- shape one is the default 'arrow' shape (set-up by *POINTER)
- to use an alternative, define and use shape 2
- when the pointer leaves the window where it was changed, it should be re-set to shape 1.

The reason codes Pointer_Entering_Window and Pointer_Leaving_Window returned from Wimp_Poll are very useful for deciding when to reprogram the pointer shape.

It is important that when selecting shape 1, you provide a shape data address of –1, so that no attempt is made to redefine the shape. On the other hand, you should always supply shape data when programming shape 2, in case that shape's definition has been changed since the last time you used it (by another program, perhaps).

Wimp_OpenTemplate &400D9

On entry: R1 = pointer to template filename to open

On exit: –

This causes the Wimp to open the file template file given, and to read in some header information from the file. Only one template may be open at a time; this is the one used by Wimp_LoadTemplate when that SWI is called.

Wimp_CloseTemplate &400DA

On entry: –

On exit: –

This closes the template file currently open.

Wimp_LoadTemplate &400DB

On entry: R1 = pointer to user buffer for template
R2 = pointer to workspace for 'indirected' icons
R3 = pointer to end of workspace
R4 = 256-byte font reference array (–1 for no fonts)
R5 = (wildcarded) name to match
R6 = position to search from (0 for first call)

On exit: R2 = pointer to remaining workpace
R6 = position of next entry (0 if no match found)
The template is at R1
The font array is updated if fonts were used
The string at R5 is overwritten by the actual name (so at least 12 bytes must be available there)

Window templates are created by the template creation utility. They are stored in a file, and each template has a name associated with it. Because the search name may be wildcarded, it is possible to search for all templates of a given form (eg 'text*') by calling Wimp_LoadTemplate with R6=0 the first time, then using the value passed back for subsequent calls. R6 will be returned as 0 when the last template is found. As the wildcarded name is overwritten by the actual one found, it must be re-initialised before every call.

The 'indirected' icon workspace pointer is provided so that when the window definition is read into the buffer addressed by R1, its icon fields can be set correctly. An indirected icon's data is read from the file into the workspace addressed by R2, and the icon fields in the window definition are set appropriately. R2 is updated, and if it becomes greater than R3, an error is given.

The font reference count array is used to overcome the problem caused with font handles. When a template file is stored, font information such as size, font name etc. is stored along with the font handle that was used to reference the font. When a template is subsequently loaded, the Wimp calls Font_FindFont using the appropriate font handle, and increments the entry for that handle in the reference array. This array should be initialised to zero.

When a window is deleted, you should call Font_LoseFont the number of times given by that font's reference count. This implies that a separate 256-byte array is needed for each template loaded. However, this can be stored a lot more compactly (eg using handle/count byte pairs) once the array has been set up using Wimp_LoadTemplate.

An alternative is to have a single reference count array for all the windows in the application, and only call Font_LoseFont the appropriate number of times for each handle when the application terminates.

Wimp_ProcessKey &400DC

On entry : R0 = character code caret data is as set up by Wimp_SetCaretPosition, or by clicking in a writeable icon

On exit : R0 = reason code (as would be returned from Wimp_Poll)

This call can be used to make the Wimp think that a given key has been pressed by the user. It is most useful in programs where a menu of characters corresponding to those not immediately available from the keyboard is presented to the user, and clicking on one of them causes the code to be entered as if typed from the keyboard.

Note that this call should only be used if the caret is inside a writeable icon, since otherwise the caret is under the control of the application, rather than the Wimp itself.

Wimp_CloseDown &400DD

On entry : –

On exit : –

From Wimp version 0.18 onwards, this call must be made immediately before the application is about to terminate (ie go back to command mode, or run the DeskTop program, or whatever). At this point the Wimp will reset the soft key settings to their original values (ie as they were before Wimp_Initialise was called), and may also not return to the application (ie if the Wimp is currently running other client applications).

Note that if a program is to be compatible with Wimp 0.17 and earlier, it must perform a series of 'bodges'; see the 'Note' above. If it is not intended to work with early Wimps, then it must specifically check for it on entry, and complain.

Error	Messages	
&280	`Wimp unable to claim work area`	RMA area full!
&281	`Unknown Wimp operation`	Invalid SWI number called
&282	`Rectangle area full`	Screen display is too complex
&283	`Too many windows`	Maximum 32 windows allowed
&284	`Window definition won't fit`	No room in internal tables
&286	`Wimp_GetRectangle called incorrectly`	
&287	`Input focus window not found`	
&288	`Illegal window handle`	You've got it wrong!
&289	`Bad work area extent`	†

† See the sections **Introduction** in the chapter **FUNDAMENTAL OS CONCEPTS** and **The output streams** in the chapter **CHARACTER OUTPUT**.

Most of these errors are provided as debugging aids to development programmers, and should not occur when the system is working properly, except for `Too many windows`, which can happen if an application program allows the user to bring up more and more windows. The error is not serious, as long as the application program's error trapping is written properly – when creating a window, any data structures relating to it should only be updated once the window has been successfully created.

The Wimp SWIs conform to the usual OS standard: it is possible to suppress error reporting when calling them. However, it is normally more sensible to install an error handler, since it is not usually possible to continue processing after an error – more often, the application should tidy up and report the error to the user.

Compatibility with Wimp 0.17 and earlier

As a result of certain incompatible changes made between Wimp versions 0.17 and 0.18, application programs wishing to be compatible with either version must invoke a series of 'bodges' to make this happen. If this is considered too much trouble, then it is acceptable for the application to complain specifically if it is asked to work with Wimp 0.17 or earlier, but it is *not* acceptable for it to refuse to work with Wimp 0.18 or later.

The bodged parts of the application program should look like this (if it is written in BBC BASIC V, that is!):

```
SYS "Wimp_Initialise" TO version%
bodgeit% = (version% < 18)
IF bodgeit% THEN
   DIM oldfx%(8)
   FOR I% = 1 TO 8:SYS "OS_Byte",I%+220,2,0 TO ,oldfx%(I%):NEXT
   SYS "OS_Byte",219,2,0 TO ,oldfx219%
ENDIF
```

create windows etc. as normal

```
REPEAT
SYS "Wimp_Poll",,block% TO action%
CASE action% OF
etc.
WHEN 8:
  key%=block%!24
  IF bodgeit% THEN
     IF key%=0 THEN key%=INKEY(0)+&100
     IF key%>=&87 AND key%<=&8B THEN
        key%=key%+&104-&10*INKEY(-1)-&20*INKEY(-2)
  ENDIF
  PROCdecodekey(key%)
etc.
ENDCASE
UNTIL FALSE
```

at point where program is about to exit:

```
IF bodgeit% THEN
     FOR I%=1 TO 8:SYS "OS_Byte",I%+220,oldfx%(I%),0:NEXT
     SYS "OS_Byte",219,oldfx219%,0
ELSE SYS "Wimp_CloseDown"
ENDIF
```

The first part of the program ensures that the soft keys return the appropriate codes, while the second part, inserted at the point where the program processes a 'key pressed' return from Poll_Wimp, translates the cursor keys into their appropriate forms. Note that it is not possible to use the *FX 4,2 setting for the cursor keys, since the Wimp itself uses the codes when decoding cursor key presses within writeable icons.

On exit, the program must reset the soft key settings to their appropriate values, otherwise the soft keys will not produce their normal effects (ie. whatever setting they had when the program was entered). On Wimp version 0.18 and later, Wimp_CloseDown will achieve the desired effect: otherwise, it must be done explicitly.

Note that Wimp_CloseDown (on Wimp versions 0.18 and later) should always be the last thing the program calls before it terminates. This is so that when the Wimp supports multiple clients, it can retain control if the task exiting is not the last task it knows about.

THE FONT MANAGER

INTRODUCTION

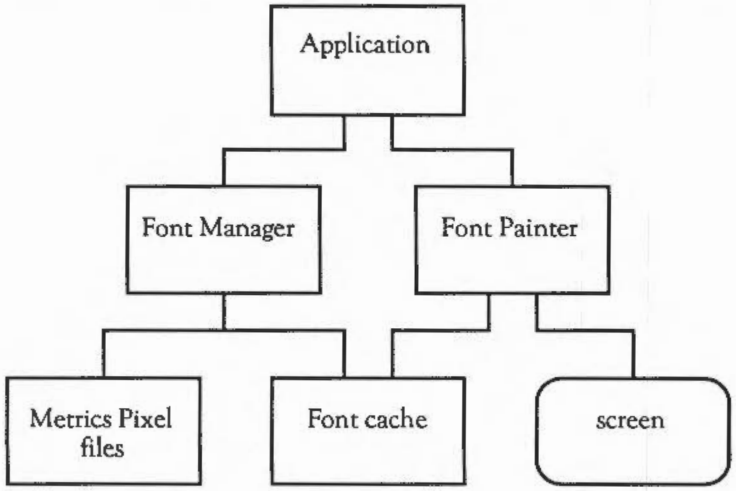
– *Note*: this chapter documents version 0.19 of the font manager.

The font system provides facilities for painting characters of various sizes and type styles on the screen.

To allow characters to be printed in any size, pixel definitions of fonts in various sizes are provided, as well as a facility to scale fonts to the desired size automatically if the exact size is not available explicitly. The fonts are, in general, proportionally spaced, and there is a facility to print justified text.

An anti-aliasing technique is used to print the characters. This technique uses shades of grey to represent pixels that should only be half filled-in. The fonts are defined with 16 shades of grey. However, they may be printed with fewer, to allow for modes with fewer colours or the use of coloured text. Text can also be printed in modes of varying resolution, by using the scaling algorithm on the pixel definitions.

The structure showing how the parts of the system fit together is given below:



The font manager

The application calls the font manager to find out information such as which fonts have been cached, how wide a particular piece of text is, and so on.

The font manager reads the data about the various fonts from disc, and is responsible for caching this data, in order to speed the process up.

The metrics of characters are held in a format independent of the output device or the size in which the font has been set up. When the metrics information is cached, the font manager scales the metrics according to the required character size, so that the numbers are held in 1/72000ths inch (ie 1/1000th of a point).

The font painter

To paint characters on the screen, the application calls the font painter. It can do this either by calling the relevant SWIs, or by means of a VDU sequence, whichever is more convenient.

Note that it is necessary for the font painter to have some kind of translation/scaling function to go from 1/72000 inch to screen external co-ordinates. This scaling factor has been fixed, by assuming that one screen unit is equivalent to 1/180th of an inch (ie there are 90 pixels per inch horizontally in mode 0).

THE FONT MANAGER

The font manager acts as an intermediary between an application and the font files, returning data such as the sizes of characters and the actual data making up the definitions. It is responsible for caching the font data, so that disc accesses will not be required for every character.

The font files

The information pertaining to a given font is held in two files; one for the metrics and one for the pixel information. These are contained in a directory whose pathname is determined by the name of the font. The components of the font name are separated by '.'s, so the directory structure can be several levels deep.

To provide some flexibility about the location of the fonts, any font access is prefixed by the system variable Font$Prefix. Hence, the font manager will access a font file by a name of the following type:

```
<Font$Prefix>.Times.Roman.IntMetrics <Font$Prefix>.Times.Roman.x90y45
```

There are two types of file in the font directories:

- Metrics files, defining the sizes of characters
- Pixel files, defining the shape of characters.

In a given font directory, there should only be one metrics file, which defines the sizes of the characters in a given font/style as ratios of the point size selected.

Different sections of the pixel file contain the same font in different sizes. When asked to cache a font, the font manager looks for the best approximation to the required size, and scales the font if necessary. Since the scaling algorithm is necessarily fast, it is not able to perform the kind of image-enhancement that the original font generation program does, so the scaled output looks slightly worse than the original 'exact' sizes.

Accessing fonts

To allow the font manager to perform automatic caching and uncaching, as well as sharing data, you must perform the following in order to access a font:

- To define the font, you specify the font name, together with the point size and screen resolution (dots per inch). The font manager returns a handle to the user, which is a number between 1 and 255 which identifies the font.

- To use the font, you specify the font handle in the relevant command.

- When you have finished with the font, you must call Font_LoseFont to tell the font manager that the font is no longer required.

When you ask for a given font, the font manager looks in its font cache to see what is already available. If the font is already present, its handle is returned to you, so in this way applications can share fonts in memory.

If the font is not available, the data is loaded from the relevant filing system, and a previously unused handle returned.

Since the font manager always knows which fonts are still in use, it can throw away any font which is no longer used. In practice, however, unused fonts are not thrown away until the font cache becomes full.

It is also possible for fonts which are still in use to be thrown away, if the font cache is too small to house all the required fonts at once. In this case, all but the font header information is deleted, so that the font can be automatically recached if required later. The font manager also knows how long ago each of the fonts was last used, so it will throw away the oldest available font if it has a choice.

To see which fonts are currently held in the font cache, type

```
*FONTLIST.
```

Font manager SWIs

When the font painter wishes to access the data cached by the font manager it can do so directly, since the two are integrated together. However, the application program is required to call the font manager in an orderly manner to read the data. An interface has been designed for this purpose.

The font manager module provides a range of SWIs from &40080 onwards, which are allocated as follows:

Font_CacheAddress &40080

On entry: R0 = 0

On exit: R0 = version number (v 0.06 and later)
R1 = amount of font cache used (bytes)
R2 = total size of font cache (bytes)

The version number returned is the actual version*100, so v. 1.07 would return 107. Versions prior to 0.06 will return 0 (ie R0 is preserved).

The call returns details about the font cache size and the amount of space used.

Font_FindFont &40081

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

On exit: R0 = font handle

This call returns a handle to a font whose name, point size and screen resolution are given.

Font_LoseFont &40082

On entry: R0 = font handle

On exit: –

This call tells the font manager that a particular font is no longer required.

Font_ReadDefn &40083

On entry:	R0 = font handle
	R1 = pointer to buffer to hold font name
On exit:	R1 = pointer to buffer (now contains font name)
	R2 = x point size * 16
	R3 = y point size * 16
	R4 = x resolution (dots per inch)
	R5 = y resolution (dots per inch)
	R6 = 'usage' count of font
	R7 = 'age' of font

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

Font_ReadInfo &40084

On entry:	R0 = font handle
On exit:	R1 = minimum x co-ordinate in pixels (inclusive)
	R2 = minimum y co-ordinate in pixels (inclusive)
	R3 = maximum x co-ordinate in pixels (exclusive)
	R4 = maximum y co-ordinate in pixels (exclusive0

This call returns the minimal area covering any character in the font. This is called the font bounding box.

Font_StringWidth &40085

On entry:	R1 = pointer to string
	R2 = maximum x offset before termination (1/72000th inch)
	R3 = maximum y offset before termination (1/72000th inch)
	R4 = 'split' character
	R5 = index of character to terminate by

On exit: R2 = x offset after printing string (up to termination)
R3 = y offset after printing string (up to termination)
R4 = no of 'split' characters in string (up to termination)
R5 = index into string giving point at which it terminated

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

If R4 contains –1 on entry, then on exit it contains the number of printable (as opposed to 'split') characters found.

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

– just before the last 'split' char reached

– if the 'split' char is –1, then before the last char reached

– if R2, R3 or R5 are not exceeded, then at the end of the string.

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

– finding the cursor position in a string if you know the co-ordinates

– finding the cursor co-ordinates if you know the position

– working out where to split lines when formatting (set R4=32)

– finding the length of a string (eg. for right-justify)

– working out the data for microspacing (as the font painter does).

Font_Paint &40086

On entry:	R1 = pointer to string
	R2 = plot type
	R3 = x co-ordinates (either OS co-ordinates or 1/72000th inch)
	R4 = y co-ordinates (either OS co-ordinates or 1/72000th inch)

On exit:	–

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

Bit	**Action if set**
0	Justify text
1	Rub-out box required
2	Absolute co-ordinates
4	OS co-ordinates supplied (otherwise 1/72000th inch)

The string is allowed to contain font-change and colour-change sequences:

,<dx0>,<dx1>,<dx>
11,<dy1>,<dy1>,<dy2>
17,<foreground colour>
17,<&80+background colour>
18,<background>,<foreground>,<font colour offset>
21,<comment string>,<terminator (any ctrl char)>
25,<underline pos>,<underline height>
6,<font handle>

After the call, the current font and colours are updated.

To provide a rub-out box, or to justify text, you must supply a bounding box by a couple of previous MOVE commands. The section on font VDU sequences explains this.

Characters 9 and 11 allow for movement within a string. This is useful for printing superscripts and subscripts, as well as tabs, in some cases. They are each followed by a

3-byte sequence specifying a number (low byte first, last byte sign-extended), which is the amount to move by in 1/72000ths of an inch.

The <underline position> following a 25 is the position of the top of the underline relative to the baseline of the current font, in units of 1/256th of the current font size. It is sign-extended by the font manager, so an underline below the baseline can be achieved by setting the underline position to a value '>' 127. The underline height determines its thickness, and is in the same units, although it is not sign-extended.

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

Font_Caret &40087

On entry: R0 = colour (EORed onto screen)
R1 = height (OS co-ordinates)
R2 = flags: (bit 4 set implies OS co-ordinates otherwise 1/72000th inch)
R3 = x co-ordinate
R4 = y co-ordinate

On exit: –

The 'caret' is a symbol used as a text cursor when dealing with anti-aliased fonts. The height of the symbol, which is a vertical bar with 'twiddles' on the end, can be varied to suit the height of the text, or the line spacing.

Font_ConverttoOS &40088

On entry: R1 = x co-ordinate (in 1/72000th inch)
R2 = y co-ordinate (in 1/72000th inch)

On exit: R1 = x co-ordinate (in OS units)
R2 = y co-ordinate (in OS units)

This call converts a pair of co-ordinates from 1/72000th inch to OS units.

Font_Converttopoints &40089

On entry: R1 = x co-ordinate (in OS units)
R2 = y co-ordinate (in OS units)

On exit: R1 = x co-ordinate (in 1/72000th inch)
R2 = y co-ordinate (in 1/72000th inch)

This call converts a pair of co-ordinates from OS units to 1/72000th inch.

Font_SetFont &4008A

On entry: R0 = handle of font to be selected

On exit: –

This call sets up the font which is used for subsequent painting or size-requesting calls (unless overridden by a 26,<font> sequence).

Font_CurrentFont &4008B

On entry: –

On exit: R0 = handle of currently-selected font
R1 = current background logical colour
R2 = current foreground logical colour
R3 = foreground colour offset (v. 0.18 onwards)

This call returns the state of the font painter's internal characteristics which will apply at the next call to Font_Paint.

The value in R3 gives the number of colours that will be used in anti-aliasing. The colours are f, f+1... f+offset, where 'f' is the foreground colour returned in R2, and offset is the value returned in R3. This can be negative, in which case the colours are f, f–1... f–offset. Negative offsets are useful for inverse anti-aliased fonts.

Offsets can range between –14 and +14. This gives a maximum of 15 foreground colours, plus one for the font background colour. If the offset is 0, just two colours are used: those returned in R1 and R2.

The font colours, and number of anti-alias levels, can be altered using Font_SetFontColours, Font_SetPalette, Font_SetThresholds and Font_Paint.

Font_FutureFont &4008C

On entry: –

On exit: R0 = handle of font which would be selected
R1 = future background logical colour
R2 = future foreground logical colour
R3 = foreground colour offset (v 0.18 onwards)

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

Font_FindCaret &4008D

On entry: R1 = pointer to string
R2 = x offset (1/72000")
R3 = y offset (1/72000")

On exit: R1 – R5 = as exit from Font_StringWidth

On exit, the registers give the nearest point in the string to the caret position specified on entry. This call effectively makes two calls to Font_StringWidth to discover which character is nearest the caret position. It is recommended that you use this call, rather than perform the calculations yourself using Font_StringWidth, though this is also possible.

Font_CharBBox &4008E

On entry: R0 = font handle
R1 = ASCII character code
R2 = flags (bit 4 set => OS coords, else 1/72000")

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

You can use this call to discover the bounding box of any character from a given font. If OS co-ordinates are used and the font has been scaled, the box may be surrounded by an area of blank pixels, so the size returned will not be exactly accurate. For this reason, you should use 1/72000th inch for computing, for example, line spacing on paper.

Font_ReadScaleFactors &4008F

On entry: –

On exit: R1 = x scale factor
R2 = y scale factor

The x and y scale factors are the numbers used by the font manager for converting between OS co-ordinates and 1/72000th inch. This calls allows the current values to be read.

Font_SetScaleFactors &40090

On entry: R1 = x scale factor
R2 = y scale factor

On exit: –

An application should set the scale factors to the desired values when it starts, as they may have been changed by a previous application. The usual value is 400 in each direction. A well-designed application reads the current value on entry and restores them when it terminates.

The following calls are available from versions 0.18 onwards only.

Font_ListFonts &40091

On entry: R1 = pointer to 40-byte buffer for font name
R2 = count (0 on first call)
R3 = pointer to directory prefix (–1 => use <font$prefix>)

On exit: R1 preserved
R2 updated (–1 if no more names)

This call searches the directory named by the variable Font$Prefix, and its sub-directories, for files ending in '.IntMetrics'. When such a file is found, the full name is put in the buffer, terminated by a carriage return. A simple program can be written using this call to list the available fonts, for example:

```
DIM buffer% 40
c%=0
REPEAT
 SYS "Font_ListFonts",,buffer%,c%,"$.fonts" TO ,,c%
 IF c%<>-1 THEN PRINT $buffer%
UNTIL c%=-1
```

The font manager command *FONTCAT performs the same function as this program.

Font_SetFontColour &40092

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

On exit: –

This call is used to set the current font (or leave it as it is), change the foreground and background colours, and offset (number of foreground colours – 1) for that font.

In 256-colour modes, the background colour is ignored, and the foreground colour is taken as an index into a table of pseudo-palette entries (see below).

Font_SetPalette &40093

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

On exit: –

This sets the anti-alias palette. The physical colours in R4 and R5 are of the form &BBGGRR00. That is, in memory they would consist of four bytes, being 0 followed by the palette entry for the red, green and blue gun respectively (see VDU 19).

In non-256-colour modes, the palette is programmed so that there is a linear progression from the colour given in R4 to that in R5, using logical colours R1, R2, R2+1... R2+R3.

In the 256-colour modes, R1 is ignored. R2 is used as an index in the range 0 – 15 of a pseudo-palette table. This table contains the closest logical colours, under the default palette, to the required physical colours given in R4 and R5. When painting takes place, these 'best fit' logical colours are used.

Also in 256-colour modes, the font offset (R5) of the font manager is used only in versions 0.18 onwards.

Font_ReadThresholds &40094

On entry: R1 = pointer to result buffer

On exit: R1 preserved, buffer contains threshold data

This call reads the list of threshold values that the font manager uses when painting characters. Fonts are defined using 16 anti-aliased levels. The threshold table gives a mapping from these 16 levels to the 2 – 16 logical colours actually used to paint the character.

The format of the data read is:

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

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

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

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

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

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

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

Font_SetThresholds &40095

On entry: R1 = pointer to threshold data

On exit: –

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

Note that some of the above calls are duplicated as VDU sequences, since in some circumstances it may be more convenient to access them this way.

The following calls are available from 0.20 onwards only.

Font_FindCaretJ &40096

On entry: R1 – R3 as Font_FindCaret
R4 = x justification offset
R5 = y justification offset

On exit: As Font_FindCaret
(they point to the nearest point where the caret can go)

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

Font_StringBBox &40096

On entry: R1 = pointer to string

On exit: R1 = bounding box minimum x (1/72000")
R2 = bounding box minimum y (1/72000")
R3 = bounding box maximum x (1/72000")
R4 = bounding box maximum y (1/72000")

This call measures the size of a string without acually printing it. The string can consist of printable characters and all the usual control sequences. The bounds are given relative to the start point of the string (they might be negative due to backward move control sequences, etc).

THE FONT PAINTER

The font painter is responsible for 'painting' characters of a given font on the screen. Requests to select a font, to paint text in a given font or to select the colour for painting are sent to the font painter by means of VDU sequences. The font painter is separate from the rest of the operating system, and communicates with the VDU queue handler via an interface involving unrecognised VDU sequences.

Converting from pixel data to the screen

The font painter must go through various conversions before outputting the text, depending on the resolution of the screen, the number of colours used for anti-aliasing and the colour of text required. You can specify some of these conversions, whilst others are performed automatically by the font painter in an attempt to overcome the limitations of the output device (the screen). For example, pixels are effectively averaged to give an approximate representation of text in a lower-resolution mode.

In summary, the text output to the screen depends on the following:

– The font and font size selected
– The resolution of the screen mode
– The number of bits used for anti-aliasing
– The colour of the text to be output.

The conversion process is done in the order above, so that text can go through several conversions before being output.

The font painter provides calls to define and select fonts, to set up the palette for anti-aliasing, and to define transfer functions for converting from four-bits per pixel to less bits.

Co-ordinate units

Since the numbers in the metrics files are held in units of 1/1000ths em, which are subsequently scaled to 1/1000 point sizes (1/72000th inch), it follows that the logical units to use throughout are 1/72000th inch. This has the advantage that rounding errors are minimal, since co-ordinates are only converted for the screen at the last moment.

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

Because of this, the font painter takes its initial co-ordinates from the user in the normal screen external co-ordinates (1280x1024 units). To make the conversion from screen units to points, the font painter assumes that there are 180 screen units to the inch (ie 90 pixels per inch horizontally in 80-column modes).

When the font painter moves the graphics point after printing a character, it does this internally to a resolution of 1/72000 inch, to minimise the effect of cumulative errors. Unfortunately, it is not possible to specify co-ordinates to this resolution, which may cause problems when trying to justify text on the screen, for example. For this reason, the font painter provides a justification facility, so the problem does not arise.

The application can obtain the widths of characters to a resolution of 1/72000 inch by calling the font manager, so its internal representation of co-ordinates can be to this resolution.

VDU sequences

Define font

- *Note*: the VDU form of Find font is included in the latest releases for compatibility with earlier versions of the font manager only. You are strongly advised to use the SWI interface in all programs; the VDU version may eventually disappear. This applies to all VDU sequences which duplicate SWI calls.

To ask the font manager for a particular font at a specified size, the application can either call Font_FindFont (see earlier), or use the following VDU sequence:

```
VDU 23,26,
        <font no>,
        <point size>,
        <x ppi>,
        <y ppi>,
        <x scale>,
        <y scale>,
        0,0,<font name>,<carriage return>
```

(The entries are shown on separate lines for clarity.) The font name is given as a string sent to the VDU immediately after the VDU 23 sequence, and is termiated by a carriage return.

Note that whereas Font_FindFont causes the font manager to choose an appropriate font handle and return it to the application, the VDU sequence specifies an absolute font handle which is to be used for that font. For this reason it is preferable to use Font_FindFont to define fonts.

<x ppi> and <y ppi>, if non-zero, specify the number of pixels per inch horizontally and vertically in the screen mode in which the font will be painted. If zero, the

values are chosen depending on the current screen mode (90 by 45 in mode 0, for instance).

<x scale> and <y scale> are provided to allow for non-integer point sizes, and also to allow the horizontal and vertical point sizes of a font to be different.

The size of the font is calculated in 1/16ths of a point, by multiplying the 'point size' by the x and y scales respectively. To specify a 12 by 10 point font, you should set the 'point size' to 1, and the x and y scales to 12 and 10 respectively. Note that the maximum value of point size, x scale or y scale is 255.

Note also, that there is no byte to set the palette or to define the transfer function, since these relate to the screen as a whole, not to each font individually. They are done instead by the VDU 23,25 sequence.

Set transfer function

The anti-aliased fonts are defined in the font files with four bits being allocated for each pixel, ie 16 grey levels are possible. When painting to the screen, however, it may be more convenient to use less grey levels, so that, for instance, text can be painted in different colours, or some of the logical colours in the palette could be used for other graphics. For this reason, the set transfer function call is provided to allow the initial 16 grey levels to be translated into eight, four or two levels:

```
VDU 23,25,<bits per pixel>,<threshold 1> ...,<threshold 7>
```

<bits per pixel> specifies the number of bits per pixel used for anti-aliasing. It must be less than or equal to four. If it is four, the rest of the bytes are ignored, as there is a one-to-one mapping from the anti-aliased colours onto the logical colours used.

The threshold values are used to decide which output bits are produced from the original values (0 to 15). There are either one, three or seven of them depending on the number of bits per pixel.

For example:

```
VDU 23,25,3,2,4,6,8,10,12,14
```

translates the values 0 – 15 to: 0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7.

Set anti-aliasing palette

The principle of anti-aliasing relies on the colours of the pixels being painted to be a linear interpolation between the 'background' and 'foreground' colours required. To facilitate the setting up of the palette registers, this call is provided to set up the relevant palette registers:

5VDU 23,25,
&80+<background logical colour>,
<foreground logical colour>,
<start R>,<start G>,<start B>
<end R>,<end G>,<end B>

<start> and <end> bytes are physical components, one for each of the red, green and blue guns (as in the VDU 19,n,16,r,g,b sequence).

The <background logical colour> is set to <start phys col>. The n–1 colours starting from the <foreground logical colour> are set to the intermediate values between the start and end physical colours. The value of 'n' is determined from the transfer function setting, and is equal to the number of colours used for anti-aliasing (2,4,8 or 16, corresponding to 1, 2, 3 or 4 bits per pixel).

As a concrete example, suppose these VDU 23s were used:

```
VDU 23,25, 2, 4, 8, 12, 0,0,0,0
VDU 23,25, &80, 4, 0,0,0, 255,255,255
```

The first command indicates that there are two bits per pixel, so four logical colours are used for anti-aliasing. The threshold values are 4, 8 and 12. Thus the sixteen anti-aliased colours will map as follows:

A – A colour	**Logical colour**	
0 – 3	background	0
4 – 7	foreground	4
8 – 1	foreground+1	5
12 – 15	foreground+2	6

The second command says that the background logical colour is 0. The red, green, blue entry (RGB) for this is 0,0,0, so the background will be black. The first logical colour for the foreground is 4, so colours 4, 5 and 6 are used for the three remaining colours. These are set to the interpolation of the physical colours between 0 and 255 on each gun. That is, the RGB components for the logical colours are set to 85, 170 and 255. Of course, if the start and end RGB components were different, then each gun would have its own interpolated value.

Several of these sequences may be required to define the palette for all text. Note that by using the same background logical colour for different sets of colours, it is possible to pack more sets of anti-aliasing colours into the palette. For example:

```
VDU 23,25,2,4,8,12|                 :REM set 2 bits per pixel output
VDU 23,25,&80, 1,  &00,&00,&00,  &F0,&00,&00
VDU 23,25,&80, 4,  &00,&00,&00,  &00,&F0,&00
VDU 23,25,&80, 7,  &00,&00,&00,  &00,&F0,&F0
VDU 23,25,&80, 10, &00,&00,&00,  &F0,&00,&F0
VDU 23,25,&80, 13, &00,&00,&00,  &F0,&F0,&F0
```

However, sharing the background logical colour in this way means that the different text colours must all have the same background colour.

Note that, because fonts are painted onto the screen by ORing them in, there is a restriction on the logical colours that can be used for the background colour, namely, that whenever a foreground colour is ORed with the corresponding background colour, it is unaffected. Normally this does not matter, since the background logical colour is usually zero, but it can be important if several sets of background and foreground colours are used. It is a much more important restriction in 256-colour modes, since it usually means that the background colour has to be black.

Set anti-aliasing palette (256-colour modes)

```
VDU 23,25,&FF,
        <logical colour>,
        <start R>,<start G>,<start B>
        <end R>,<end G>,<end B>
```

Because the 256-colour modes have been set up to use a specific palette, the font painter should not change the actual palette in these modes as it would in the others.

Instead, a special call is provided for these modes. To facilitate changing colours within painted text, the font painter maintains a 'pseudo-palette' of 16 items, each of which can be set up individually by using the above call, with <logical colour> indicating the entry which is to be set up.

For each of the 12-bit physical colours that the font painter requires to set up for anti-aliasing, a translation occurs that attempts to select the closest colour available under the default palette. The top two bits of each gun can be represented exactly using this palette, but the bottom two bits of the output colour represent the tint value, which is an amount of white to add to the colour. The formula used to compute the closest approximation to the required colour is as follows:

tint value = (2*green + red + blue) / 4

where red, green and blue are the lowest two bits of the red, green and blue guns respectively.

Paint characters

This call is equivalent to the Font_Paint command detailed earlier, and is used to display a string of anti-aliased characters on the screen. The string can contain font changes and colour changes, along with automatic justification and rub-out:

```
PLOT &D0..&D7,<x co-ordinate>,<y co-ordinate> <textual string>
```

The least significant three bits of the plot code have the following meanings:

Bit	Meaning
2	Relative/absolute co-ordinates (as normal)
1	Rub out background as well (use previous cursors)
0	Justify text within a box (use current cursor)

Note that if rub out and justify are both specified, then justify uses the current and new graphics cursors, whilst rub-out uses the old cursor and the one before that. If rub-out only is specified, then the rub out box is determined from the current and old cursor positions.

For example:

```
MOVE 0,0 : MOVE 1000,100          : REM set up rub-out box
MOVE 800,40                       : REM set up justification
PLOT &D7,200,40:PRINT "Hello"     : REM paint text

MOVE 0,0 : MOVE 1000,100          : REM set up rub-out box
PLOT &D6,200,40:PRINT "Hello"     : REM paint text
```

The <textual string> is terminated by any character less than ASCII 32, unless it occurs in one of the control sequences documented above in SWI Font_Paint.

For example, consider the following program segment:

```
VDU 23,26,1,14|:PRINT "Times.Roman"          :REM 14 point Times.Roman
VDU 23,26,2,14|:PRINT "Times.Italic"         :REM 14 point Times.Italic
VDU 23,25,3,2,4,6,8,10,12                    :REM 3 bits per pixel
VDU 23,25,&80,1,&00,&F0,&F0,&00,&00,&F0      :REM blue on cyan
VDU 23,25,&88,9,&E0,&60,&80,&F0,&00,&00      :REM red on pink

MOVE 0,0 : MOVE 1000,100                     :REM set up rub-out box
PLOT &D6,100,30                              :REM start painting sequence
VDU 26,1:PRINT "Some Times.Roman text, ";
VDU 26,2:PRINT "and some Times.Italic ";
COLOUR &80:COLOUR 1:PRINT "in blue";
COLOUR &88:COLOUR 9:PRINT " and in pink"
```

Some points to note:

- The font painter will not do anything on the screen until the final <cr> is sent, so the text is painted (and the background rubbed out) in one go.
- When a call is made to set up the palette, the current text colour is also set up to use it, so the first part of the string is painted in red on a pink background.
- When repainting a portion of a line of text, it is not sufficient to plot the text starting at the point on the screen that the portion appears to be at. As a result of the fact that the font painter holds the text co-ordinates to a higher resolution than the screen co-ordinates, the 'phase' of the internal counter is different, causing certain characters to be printed in a slightly different position than last time. The solution is to repaint the entire line of text, having set a graphics window to cover the part that you are expecting to change.

THE FONT FILES

The font files are held in a machine-readable form rather than as text, for the following reasons:

- The files are smaller, allowing more fonts to be supplied
- The data is easier for the font painter to handle
- It is a simple matter to translate the files into this form.

The format of these files is as follows:

Format of the metrics files

For any given font in a particular style, there is only one metrics file. The dimensions held in the metrics file are in 1/1000ths em, so to obtain the value in points, multiply by the point size of the font.

The format of the internal metrics file is as follows (offsets are in bytes):

Size	Offset	Meaning
40	0	Name of font, terminated by <cr>
4	40	16
4	44	16
1	48	n = number of defined characters
3	49	reserved
256	52	character mapping (ie. indices into following arrays)
2n	308	x0 – bounding box of character (in 1/1000ths em)
2n	308+2n	y0 – co-ordinates are relative to the 'origin point'
2n	308+4n	x1 – 2 bytes per entry
2n	308+6n	y1 –
2n	308+8n	x-offset after printing this character
2n	308+10n	y-offset after printing this character
	308+12n	

The character mapping is used to conserve space, since it is assumed that not all of the characters will be defined. Any character not defined should have an index of zero in the character mapping, and the first item in each of the arrays should contain data for the 'undefined' character.

This arrangement does save space as long as 260+12n < 12x256, ie. n < 235. Note that it also allows data to be shared between characters if appropriate, thereby saving more space.

Format of the pixel files

The pixel files hold the actual pixel-by-pixel definition of each of the characters in a particular font. This varies with the point size of the font, so the pixel file holds data for a font in a variety of sizes. Not all sizes are defined explicitly, but the font painter is able to generate intermediate sizes itself (with slight reduction in quality).

Index

Size	Offset	Meaning
1	0	point size
1	1	bits per pixel (ignored)
1	2	pixels per inch (x-direction)
1	3	pixels per inch (y-direction)
4	4	check sum
4	8	offset of pixel data in file
4	12	size of pixel data (needed as blocks may not be consecutive).
	...	more of the same 4-word blocks (as many as you like)

The index is terminated by a zero byte.

Pixel data: (word-aligned)

Size	Offset	Meaning
4	0	x-size in 1/16ths point * pixels per inch (x)
4	4	y-size in 1/16ths point * pixels per inch (y)
4	8	pixels per inch (x-direction)
4	12	pixels per inch (y-direction)
1	8	x0 – maximum bounding box for any character
1	9	y0 –
1	10	x1–x0 – ie the width in pixels
1	11	y1–y0 – ie the height in pixel rows
512	12	2-byte offsets from table start of character data (pixel data in file is limited to 64K per block)

Character data: (word-aligned)

Size	Offset	Meaning
1	0	x0 – x0,y0,x1,y1 is the bounding box (in pixels)
1	1	y0 – (signed, ie –128 to 127)
1	2	x1–x0 – ie the width in pixels
1	3	y1–y0 – ie the height in pixel rows
	4	each nibble = pixel colour (rows from bottom to top) number of nibbles = (x1–x0)*(y1–y0) word-aligned after pixel nibbles (rows are not necessarily byte-aligned)

Note that the format of the pixel data may be different if a different number of bits is used per pixel. The same principle applies, however – the data is compacted, and word-aligned at the end.

Format of the font cache

The first 5K of the font cache consists of the following tables, which act as an indices into the cache area:

Size	Offset	Meaning
&400	&0000	pointers to metrics information (scaled to font size)
&400	&0400	pointers to pixel definitions
&400	&0800	sizes of metrics information
&400	&0C00	sizes of pixel definitions
&400	&1000	table of ages, usage and 'uncached' flags

Note that the entry for font 0 is unused (since 0 is not a valid font number).

The format of the metrics and pixel data in the font cache is similar to that in the files, but with some important differences:

Format of metrics data in the font cache

Size	Offset	Meaning
40	0	name of font (terminated by <cr>)
4	40	x-size of font (in 1/16ths of a point)
4	44	y-size of font (in 1/16ths of a point)
1	48	n (= number of defined characters)
3	49	reserved
256	52	character map
4n	&134	x0 –
4n	&134+4n	y0 – bounding box (words)
4n	&134+8n	x1 –
4n	&134+12n	y1 –
4n	&134+16n	x-offset after printing this character
4n	&134+20n	y-offset after printing this character

Note that the numbers stored in the font cache have been multiplied by the point size of the font to give the appropriate values, and that they are stored in four bytes each, rather than two bytes as they are in the internal metrics files.

Format of Pixel Data in the font cache

Pixel data: (word-aligned)

Size	Offset	Meaning
4	0	point size * 16 * x-resolution
4	4	point size * 16 * y-resolution
4	8	x-resolution
4	12	y-resolution
1	16	x0 – maximum bounding box for any character
1	17	y0 –
1	18	x1–x0 – ie the width in pixels
1	19	y1–y0 – ie the height in pixel rows
8	20	Offset/size (words) in file of first 32-char chunk
8	28	Offset/size of second 32-character chunk in file

8	76	Offset/size of eighth 32-character chunk in file
4	84	Start of character data for 1st chunk (0 => not cached)
4	88	Start of character data for 2nd chunk
4	112	Start of character data for 8th chunk

The first two words indicate the scaling factor that is relevant to the font scaling algorithm. The factor of 16 is due to the way in which the 'define font' VDU sequence is specified, you can ask for a non-integer point size by changing the x or y 'scale factors' from their default values of 16.

Character data: (word-aligned)

Size	Offset	Meaning
1	0	x0 – x0,y0,x1,y1 is the bounding box (in pixels)
1	1	y0 – (signed, ie –128 to 127)
1	2	x1–x0 – ie the width in pixels
1	3	y1–y0 – ie the height in pixel rows
	4	each nibble = pixel colour (rows from bottom to top) number of nibbles = (x1–x0)*(y1–y0) word-aligned after pixel nibbles (rows are not necessarily byte-aligned)

SOUND

INTRODUCTION

The Sound System software is designed to provide a real-time polyphonic audio signal synthesis and playback system which is able to support music and speech applications and to provide simple sound effects. The overall objective in the design of the system software has been to provide features which allow the processing power of the A-series machines to be exploited in complex real-time sound synthesis, a feature normally only attainable with custom hardware.

The special purpose hardware provided on ARM-based systems simply provides a DMA output channel and analogue output circuitry to allow up to eight independent high quality stereo imaged sound or music voices to be output at a programmed sample rate; filters and mixing circuitry are provided on the main board, and both a stereo output (suitable for driving personal hi-fi stereo headphones directly, or connecting to an external hi-fi amplifier) and monophonic output to the loudspeaker are built into the system.

All sound synthesis is accomplished in software; the harmonic content and amplitude envelope characteristics for a voice are entirely programmable. A sign and logarithm number system is used in order to allow the ARM CPU to execute simple signal processing algorithms efficiently; thus complex operations such as Frequency Modulation algorithms are possible entirely in software.

The Sound System software is segmented into three levels; a range of synthesis algorithms can be loaded as Relocatable Modules to support music and speech applications, and the layered structure allows powerful expansion and customisation for particular applications.

Features of the sound system include:

- The power and speed of the ARM processor, which is exploited to make sound synthesis possible entirely in software.
- One, two, four or eight independent audio channels can be supported, each with associated programmable stereo image position.

- Programmable data sample rates with high-quality audio sampling at a 20kHz default rate per voice but with the capability to support rates up to 32kHz per voice.

- Support for Sound Voice Generators. High-speed synthesis algorithms provide streams of data samples (which can be shared by one or many sound channels). Standard built-in generators are provided for simple tone generation, and filtered wavetable algorithms for plucked string or percussive effects. External voice generator interface provided for external user-supplied algorithms.

- Applications Interface, the system interface as seen by the applications programmer. Facilities are provided for:

- Sound system configuration. (Sample rates, Channel allocation, etc.)

- Event Queue and Synchronisation facilities.

- Voice assignment and allocation.

SOFTWARE STRUCTURE

The sound system is envisaged as a hierarchical layered structure in which the raw DMA system provided by the Video Controller activates services in the higher layers to generate blocks of sound data samples for output to the Digital to Analogue Convertor. Given that the sound system requires all tone generation to be performed in software (whether raw mathematical synthesis or simpler interpolation of precompiled tables) considerable attention must be given to the demands on processor bandwidth assigned for synthesis for more powerful or complex sound production. Rather than providing built-in operating system support for very complex sound generation directly, the approach taken provides a number of levels of support with 'hooks' available to external applications programs which wish to perform especially complex sound synthesis. Usually, the system level interfaces would be used to generate sounds using one of a number of provided synthesis algorithms.

The three main levels of sound system software are briefly summarised below and then described in depth in the following sections.

SoundDMA (Level 0) – the sound DMA buffer handler

The DMA Buffer Handler is activated every time a new buffer of sound samples is required, and is responsible for manipulating the physical address pointers to the blocks of memory that the memory controller cycles through at the programmed sample rate. Level 0 is the essential service that has to be provided in firmware requiring privileged supervisor status to program the system devices.

This level of the sound system basically implements a double-buffered output channel. Level 0 activates the higher levels of sound system software to cause the next DMA buffer of sound samples to be filled; because buffer filling demands fairly large amounts of processor bandwidth, the Voice Generators have to run interruptable. All code must be resident in main memory (must be directly executable in order to fill in real-time).

Level 0 provides the facility to handle sound system overload, when the peak demands of many complex voice algorithms exceed the available processor bandwidth. (This is a function of the video mode, IRQ and FIQ bandwidth.) The offending channel is marked as Overrun and the real-time buffer filling is aborted and restarted.

Level 0 also performs all the hardware-dependent programming services for the applications programmer which require privileged-mode access. The stereo image position registers in the Video controller, and the built-in loudspeaker enable interfaces are both provided, as well as the master Audio On/Off control which disables or enables the entire sound system DMA and Interrupt system.

SoundChannels (Level 1) – the sound channel interface

The sound DMA hardware in the video controller allows programming of both sample rate and number of stereo positioned channel outputs. The physical channels imply the way in which data samples must be interleaved throughout the buffers which must be built up. The number of physical channels is constrained to one, two, four or eight; however, Level 1 manages any unassigned channels directly. A simple unified interface to whatever voice is assigned to a particular channel is provided which allows direct real-time control of musical parameters.

Channels are allocated in descending priority order, and may be de-activated by Level 0 if the time-out period for real-time buffer filling is exceeded. Level 1 provides automatic Channel flushing as channels are allocated and deallocated.

This level also provides the system services to Voice Generators: internal tables are built and maintained for both volume and musical pitch, and the interface to allow you to attach channels to installed voice generators. The system-maintained volume, in fact, maintains internal linear and logarithmic lookup tables which Voice Generators would normally use in order to scale their amplitude to the current volume setting.

SoundScheduler (Level 2) – sound event scheduler

The Sound Event Scheduler provides a time-ordered event queue manager and data structures to allow multiple channel music or sound to be produced and synchronised under simple program control. Notes, timbral changes or even user-supplied code routines may be scheduled in arbitrary time order. These are activated, as events, at the appropriate tempo-dependent time in the future. This level provides the system-maintained services for both tempo and beats (per bar); events may be queued relative to the start of a bar (or relative to the last event schedule period), and tempo may be dynamically changed whilst maintaining note synchronisation.

Sound voice generators

Each sound channel must be assigned a particular Voice Generator. The sound system Level 1 software provides interfaces to install and remove loadable Voice Generator modules. These essentially generate the next series of signal data samples requested by the sound channel handler (as well as provide entry points to start up, and close down the channel). At this level the efficiency of the code for each of the generators is paramount; wavetable synthesis can be performed very efficiently by the ARM processor once all the working variables have been loaded up into registers, and is well suited to fast sequential buffer filling. An interface is again provided at this level to external user-supplied generators which may be called up by the channel handler. The library of internal generators includes:

- simple waveform oscillators
- a speech data interpolator
- simple plucked-string and percussive table-filtering algorithms.

More complex facilities such as frequency modulation could be provided if implementation of the more essential features allows time for such developments.

Each Voice Generator would normally be allocated in resident System Heap space and may provide workspace for one or more instantiations in order to allow one or more channels to use the algorithm.

SOUNDDMA (LEVEL 0) – SOUND DMA BUFFER HANDLER

The sound system DMA hardware basically requires two (or more) physical buffers in main memory. Two sets of START/END register pairs are provided in the sound DMA Address Generator (DAG): the CURRENT set provides the start and end points through which the sound DMA pointer increments, whilst the NEXT start and end are available to the programmer to set up once the buffer-filling software has filled the new sound sample data buffer.

The DMA buffer handler is entered upon an interrupt from the IO Controller indicating that the DAG sound pointer in the Memory Controller has switched physical memory buffers. (The programmed NEXT start and end become the current values and the original pair are scrapped ready for reprogramming.) It should be noted that if the buffer-filling software does not fill and make available the next buffer whilst the DMA occurs, then the Interrupt Service Routine is re-entered. In order to fix up re-entrancy, and to mark as overrun the channel which failed to fill in time, a semaphore is maintained and a mark left on the IRQ mode Stack. So on re-entrancy:

- the known stack contents are used to determine which channel did not fill in time:
- the previous Level1 is aborted
- the offending channel marked as overrun
- the next DMA buffer fill initiated.

This prevents the situation where the DMA pointer recycles within the CURRENT start/end addresses which causes the last chunk of sound samples to be re-output, resulting in bad audible discontinuities.

Level 0 * commands

Commands are provided as follows:

***AUDIO**

The *Audio command controls the sound system.

Syntax: *Audio ON | OFF

Turning Audio Off silences the sound system completely, and stops all Sound Interrupt and DMA activity. Turning Audio back on restores the sound DMA and interrupt system to the state it was in immediately prior to turn-off. All Level 1 and Level 2 activity is effectively frozen during the time the Audio system is off, but software interrupts to all levels are still permitted, even if no sound results!

***SPEAKER**

The *Speaker command controls the loudspeaker.

Syntax: *Speaker ON | OFF

This command only affects the internal loudspeaker built into the system and not the external stereo headphone/amplifier output on the machine. It mutes the monophonic mixed signal to the internal loudspeaker amplifier.

***STEREO**

*Stereo sets the stereo position of a sound channel.

Syntax: *Stereo <channel> <position>

<channel> is 1 – 8
<position> is –127(full left) to +127(full right), 0 for centre

This command sets the stereo image position of a sound channel.

Level 0 SWI calls

A series of software interrupt service entries are provided which allow you to configure and extend the layered sound system.

Sound_Configure – Configure the sound system

This software interrupt is used to configure the number of sound channels, the buffer sample period and the number of sound samples per buffer for each channel (and for specialised applications to replace the Level 1 and Level 2 handlers which are called by Level 0). All current parameters may be interrogated by using zero input parameters, and the actual values programmed are, in fact, subject to a number of criteria such as DMA buffer length and minimum and maximum sample periods.

On Entry: R0 = no. of channels (n) rounded up to 1,2,4 or 8 (N)
R1 = samples per buffer
R2 = uS per sample
R3 = Level1 Handler (normally 0 to preserve system Level 1)
R4 = Level2 Handler (normally 0 to preserve system Level 2)
(all parameters 0 for don't change)
Constraints:
1 <= R0 <= 8 (rounded up to 1, 2, 4 or 8)
16 <= R1 * N <= Sound DMA Buffer Limit
3 <= R2 * N <= 255

On Exit: previous R0,R1,R2,R3,R4

– *Note*: multiple channels are built up by multiplexing channels into what is effectively one half, one quarter or one eighth of the sample period, and thus the signal level per channel is scaled down by the same amount. Thus the overall signal peak level for all multi-channel modes is the same, but the signal level per channel is scaled. Level 1 software could, of course, scale amplitudes for the one, two and four channel modes down to the the same perceived loudness, at the expense of signal-to-noise.

Sound_Enable – Sound system control

This software interrupt is used to enable or disable the DMA and Interrupt requests from the system. This guarantees to inhibit all sound system bandwidth consumption once a successful disable has been completed.

On Entry: R0 = new state:
bit 1 is one to update state
bit 0 is 1 for ON, 0 for OFF
(0 for don't change)

On Exit: R0 = previous state
3 for active On
2 for closedown in progress
1 for closedown imminent!
0 for OFF

SWI Sound_Speaker – Loudspeaker control

This software interrupt is used to enable or disable the internal loudspeaker channel. The stereo output socket is always enabled but the mono-mix of the left and right audio channels is switchable to the built-in amplifier and loudspeaker.

On Entry: R0 = new state:
bit 1 is one to update state
bit 0 is 1 for ON, 0 for OFF
(0 for don't change)

On Exit: R0 = previous state
1 for ON
0 for OFF

Sound_Stereo – Set stereo image position

This software interrupt is used to program the stereo image position. Depending on the number of physical channels programmed, this software call may be used to program the position of a logical channel (the normal use) or may be used for special effects to map interleaved samples for one channel to alternate positions.

On Entry: R0 = channel (C) to program
R1 = image position:
- 0 is centre
- 127 for max right
- –127 for max left
- –128 for don't change (read previous)

On Exit: R0 preserved
R1 = previous image position

For N physical channels enabled, this call will program stereo registers C, C+N, C+2N up to image reg 8.

To help clarify the above, if four channels are currently in use, and it is desired to program channel 2 as a 'stereo' voice, then programming must be performed as follows:

```
; if it is desired to preserve old slot 6 then:
  MOV R0,#6
  MOV R1,#0
  SWI Sound_Stereo
; then re-program as follows:
  MOV R1,#pos
  SWI Sound_Stereo ; programs slot 2,6 (returns old slot 2)
  ADD R0,R0,#4
  MOV R1,#altpos
  SWI Sound_Stereo ; programs slot 6
```

This Software call only updates RAM copies of the stereo image registers and the new positions, in fact, take effect on the next sound buffer interrupt. (The software interrupt may be called from IRQ code directly for scheduled image movement.)

Operational considerations

DMA buffer size and sample rate

The length of buffer is an important consideration from two points of view. The sample rate and number of channels interact together to consume sound sample bytes at a variety of rates, and the resulting buffer request interrupt rate must be chosen such that the processor is interrupted at a 'sensible' rate. A default buffer period of the order of a centi-second is selected as a value which allows reasonable temporal resolution of note lengths. Conceivably, some user tasks might wish to work with longer blocks (i.e. replaying pre-compiled speech or music from disc) so the length parameter should be extendible.

The sample rate constraints arise because of the DMA request conflicts that arise from the Video Controller when operating at high resolution screen modes. This imposes a limit of a 4 microsecond minimum sample period to DMA correctly with all screen modes, and would occur when all eight hardware channels are enabled. Outputting a byte to one of eight channels every 4 microseconds results in an audio sample period of 32 microseconds per channel, a sample frequency of 31.25kHz.

The current system design revolves around a VIDC system clock of 24MHz; however, experimental hi-resolution monochrome systems using a 28MHz VIDC clock, and systems with slow (150nS) memory which could use a 20MHz crystal, constrain the Sound sample rates (the sound clock is basically a multiple of 1/24 of the VIDC system clock). Analysing clock division rates for all these three clock options reveals that there is only one simple common denominator which enables the sample rate to default to the same value (eg important for speech sample rate interpolation). This turns out to be 6uS as follows:

6uS Sound Frequency Generator period:
5 at 20MHz
= 6 at 24MHz
= 7 at 28MHz

Thus this is chosen as the system default:

8 channel multiplex rate = 166.666 kHz
Overall audio sample rate = 20.833 kHz
resulting Nyquist = 10.416 kHz

Returning to the DMA buffer length, this is constrained to be a multiple of 4 words, and to give a buffer period of 1 centi-second, buffer sizes as outlined below are closest:

Buffer lengths for Audio Sample Rate of 20.833 kHz

	DAC o/p		
	0.9984 csec	1.0752 csec	period/byte
1-channel	208 bytes	224 bytes	48 usec
2-channel	416 bytes	448 bytes	24 usec
4-channel	832 bytes	896 bytes	12 usec
8-channel	1664 bytes	1792 bytes	6 usec
bytes per channel	&D0	&E0	
interrupt rate	100.16 Hz	930.1 Hz	

The system default buffer period is chosen as 0.9984 centi-seconds, thus the buffer length is a multiple of 52 words (13 DMA quad-word cycles).

DMA Buffer Format

The sound DMA system systematically outputs bytes at the programmed sample rate; each (16-byte) load of DMA data from memory is synchronised to the first stereo image position. For single channel operation all stereo image registers are mapped to the same position, and the buffer format is simply sequential sample bytes.

Multiple channel operation is possible with two, four or eight channels; in this case the data bytes for each channel must be interleaved throughout the DMA buffer at 2, 4 or 8 byte intervals.

Schematically:

Single channel operation:

sample rate = 20 kHz
image registers 0 – 7 programmed identically

0	byte0	byte1	byte2	byte3	byte4	byte5	byte6	byte7
+8	byte8	byte9	etc					

Two channel (a/b) operation:

sample rate = 40 kHz
image registers 0,2,4,8 and 1,3,5,7 programmed per channel

	a	b	a	b	a	b	a	b
0	byte0	byte0	byte1	byte1	byte2	byte2	byte3	byte3
+8	byte4	byte4	etc					

Four channel (a/b/c/d) operation:

sample rate = 80 kHz
image registers 0+4, 1+5, 2+6, 3+7 programmed per channel

	a	b	c	d	a	b	c	d
0	byte0	byte0	byte0	byte0	byte1	byte1	byte1	byte1
+8	byte2	byte2	etc					

Eight channel (a/b/c/d/e/f/g/h) operation:

sample rate = 160 kHz
image registers 0,1,2,3,4,5,6,7 programmed per channel

	a	b	c	d	e	f	g	h
0	byte0	byte0	byte0	byte0	byte0	byte0	byte0	byte0
+8	byte1	byte1	etc					

The interleave is entirely a function of the Level 1 channel handler which is activated every time buffer filling is required.

The system provides two DMA buffers, and performs all the physical memory DMA memory addressing, and virtual memory mapping to the higher level Sound System modules.

Building a Level 1 sound channel handler

The Level 1 interface is provided by a Control Block, a pointer which must be registered with Level 0 using the Sound_Configure interface. The first two word entries (which must be word-aligned) are defined as pointers to the Level1 handler code, and a Level1 overrun fix-up entry which is called if the previous Level1 filling has not completed in time.

Level 1 control block

```
FillCodePtr  = 0   ; pointer to Level1 Handler
FixUpCodePtr = 4   ; pointer to Level1 overrun code
 ...
ChanPtrBase  =
Chan1Ptr      = ChanPtrBase
Chan2Ptr      = ChanPtrBase + 4
Chan3Ptr      = ChanPtrBase + 8
Chan4Ptr      = ChanPtrBase + 12
Chan5Ptr      = ChanPtrBase + 16
Chan6Ptr      = ChanPtrBase + 20
Chan7Ptr      = ChanPtrBase + 24
Chan8Ptr      = ChanPtrBase + 28
1
```

Example Level 1 handler:

```
 .Level1Fill ; simple Level 1
 ; entered indirected through LDR PC,[R9,#FillCodePtr]
 ;
 ;  R6 is -ve if updated level0
 ;  R8 = sample period in uS
 ;  R9 = pointer to Level1 Control Block
 ; R10 = DMA Buffer End (+1)
 ; R11 = DMA Buffer Inc
 ; R12 = DMA Buffer Base
 ; R13 = IRQ stack pointer
 ; R14 = return to Level 0
 ; USAGE:
 ;  R7 = Channel counter
 ;  R0-5 (AND preserved R6-12 for Channel fill)
    STMFD R13!,{R14}
    ADD   R9,R9,#Chan0Ptr
    MOV   R7,#0
 .Level1Loop
    ADR   R0,Level1Return  ; return address
```

```
   STMFD R13!,{R0,R6-R12}
;; {R0} return address MUST BE LAST ON STACK!
   ADD   R12,R12,R7         ; R12 is interleaved Channel dma base
   LDR   R0,[R9,#4]!        ; call next chanel fill
   ORRS  PC,R0,#&00000002 ; re-enable IRQs in fill code
;; MUST return with LDMFD R13!,{PC}
.Level1Return
   TEQP  PC,#&08000002      ; disable IRQS again
   MOV   R0,R0              ; wait for IRQ bank
   LDMFD R13!,{R6-R12}
   ADD   R7,R7,#1           ; sequence through channels
   CMP   R7,R11
   BLT   Level1Loop
   LDMFD R13!,{PC}          ; return to Level0
;***********************************
.Level1Fixup
;On Entry:
; r14 is return
; r12 is stack mark (TOP) ** THIS MUST BE PRESERVED **
; r11 is Level1 base (code entered at  base + 4)
; stack (words) contains:
;       R10   )                   <---- TOP-1
;        R9   )
;        R8   )
;        R7   )
;        R6   )     Level 0
;        R5   ) reg save area
;        R4   )
;        R3   )
;        R2   )
;        R1   )
;        R0   )                  <---- TOP-11
;----------------------
;     return  to Level 0     <---- TOP-12
;     level 1 save block
;     (stacked R7 will contain
;       R12   DMA Base                  -13
;       R11   DMA Interleave            -14
;       R10   DMA Limit                  -15
;        R9   Sound Stream Base          -16
;        R8   Sample rate period         -17
;        R7   Channel counter            -18
;        R6   Update (flush) flag        -19
;        Stream Link return              -20
;----------------------
;   any currently stacked registers
;----------------------
;     int return          new sound IRQ entry!
;       R12
;       R11                 <------ R13
;----------------------
;
   STMFD   R13!,{R11,R12,R14}
   LDR     R14,[R12,#-18*4] ; get offending Channel no.
; disable or mark the problem Channel,
; Level0 will patch the stack, swap buffers
; and call Level1Fill imminently
   LDMFD   R13!,{R11,R12,PC}
```

Building a Level 2 sound scheduler

The Level 2 interface is less complex that outlined for Level 1, and is registered with Level 0 using the Sound_Configure interface. The first word of the block of workspace (which must be word-aligned) referenced by the pointer, is defined to be a pointer to the Level 2 handler code. The Level 2 handler code is called every DMA buffer fill period and, assuming Level 2 is implemented as a relocatable module, sets R12 to point to the workspace base.

Level 2 control block:

```
SchedulerCodePtr  = 0   ; pointer to Level1 Handler
 ...
```

Example Level 2 handler:

```
 .Level2 ; simple Level 2 skeleton
 ; entered indirected through LDR PC,[R12,#SchedulerCodePtr]
 ;
 ; R12 = Level 2 (workspace) control block pointer
 ; R13 = IRQ stack pointer
 ; R14 = return to Level 0
 ; USAGE:
 ;  R8-R11 are already preserved on entry
 ;  R0-7 MUST be preserved by the user
 ; in IRQ node, with IRQ's disabled
    STMFD R13!,{R0-R7,R14}

 ; scheduler code

    LDMFD R13!,{R0-R7,PC}
/}
```

SOUNDCHANNELS (LEVEL 1) – SOUND CHANNEL HANDLER

The channel handler accepts a buffer fill request with a virtual memory base address and length, and converts this to a series of Voice Generator calls with selected buffer interleave offsets to build up the required composite sound buffer.

The Sound Channel Control Block (SCCB) contains the relevant parameters to map the buffer to the one to eight active sound channels. The hardware supports only one, two, four or eight physical channels, so in order to support an intermediate number of channels the next highest physical number is chosen, and the unused channels must have their interleaved data cleared to zero amplitude. If this is performed at channel initialisation, then no further work in software is required to maintain these unused channels.

Level 2 * commands

Commands are provided as follows:

***VOLUME**

*Volume command sets the audio channel loudness.

Syntax: *Volume n <range 1 – 127>

Scales the internal lookup tables that the voice generators would normally use, to scale their signal amplitude efficiently.

***VOICES**

*Voices lists the installed voice generators and channel allocation.

Voice Generators, when they are installed, are entered into a system-maintained table and may be attached to one of the eight possible sound channels. It is possible to attach all eight channels independently of the number of channels currently enabled in Level 0.

***CHANNELVOICE**

*ChannelVoice command attaches a Voice Generator to a Sound Channel

Syntax: ChannelVoice <channel> <voice index> | <voice name>

This command is used to configure the allocation of voice channels to voice generators. The voice index is the number given by the *Voices command (and an index of 0 may be used to mute the channel altogether). The voice index may well depend on the order in which the voices were loaded into the system, the (unquoted) voice name allows position independent voice attachment. However, this is only supported on SoundChannels version 1.08 upwards. The voice name is case sensitive and only an exact match of names will result in the channel voice attachment being successful.

*SOUND

The *Sound command makes a foreground (immediate) sound.

Syntax: *Sound <channel> <amplitude> <pitch> <duration>

This command provides the equivalent of the BASIC SOUND command and allows immediate sounds to be generated on the specified channel, providing the channel number is in fact active according to the current number of channels enabled. The parameters must all be unsigned integers and are interpreted identically to the SOUND command.

*TUNING

The *Tuning command sets the system tuning.

Syntax: *Tuning n <range 1 – 32767>

This command overrides the system pitch base. It is not to be supported in the same way in future; instead, a signed representation of relative pitch change will be implemented. All pitches for Voice Generators vary with the system pitch base.

*CONFIGURE SoundDefault

This sets default sound channel one parameters.

Syntax: SoundDefault <0 | 1> <0 – 7> <1 – 16> (speaker, coarse volume, voice)

This configure option sets the non-volatile parameters to be used after power-on. They specify the built-in loudspeaker as on or off, the relative default volume preferred at start up, and the voice generator one wishes to attach to channel 1 (the default system Bell channel).

The volume parameter is a 'coarse' volume setting, each unit corresponding to one eighth of the overall maximum volume. At power-up the default settings are: one channel, with central stereo position.

Level 1 SWI calls

A table of installed Voice Generators is managed by Level 1. Services are provided to allow voices to be installed and removed from the Voice Generator Table, and to attach particular voices to channels. To simplify the software interface to user-loadable voices, facilities are provided to determine textual voice names and channel assignment.

Sound_InstallVoice – Install voice generator

This software interrupt is used by Voice Modules or Libraries to associate a RAM-resident Voice Generator with a Sound Voice Generator entry. A Voice Generator must have a header of pointers to code and data fields as specified in the section **Sound voice generators** for the SoundScheduler (level 2).

On Entry: R0 = Voice Module pointer (0 for don't change)
R1 = voice slot specified (0 for install in next free slot)

On Exit: R0 = string pointer – name of previous voice (or error message)
R1 = voice number allocated (0 for FAIL to install)

This call may be used to interrogate the installed voice list (by using an R0 parameter of 0 for each slot entry); the voice table is currently limited to 32 entries.

Sound_RemoveVoice – Remove voice generator

This software interrupt is used when Voice Modules or Libraries are to be removed from the system. It notifies Level 1 that a RAM-resident Voice Generator is being scratched. (It may also be called when the Relocatable Module Area is Tidied).

On Entry: R1 = voice slot to remove

On Exit: R0 = string pointer – name of previous voice (or error message)
R1 is voice number de-allocated (0 for FAIL)

Sound_AttachNamedVoice – Attach a channel to a named voice generator

This call is used to bind a particular sound channel to one of the loaded voices. The name is used as the parameter, and if no matching loaded voice name is found then an error is reported and the channel is not detached from the voice to which it was previously bound. An exact case-sensitive character match is performed.

On Entry: R0 = channel number (1 – 8)
R1 = pointer to Voice name (ASCII string, 0 terminated)

On Exit: R0 is preserved, or 0 for FAIL
R1 is preserved

Attaching a new voice results in the previous voice being shut down and the new voice being reset. Different algorithms have different internal state representations so it is not possible to swap Voice Generator mid-sound.

SWI Sound_AttachVoice – Attach channel to voice generator

This software interrupt is used to attach an Installed Voice Generator with a channel number.

On Entry: R0 = channel number (1 – 8)
R1 = voice slot to attach (0 to detach and mute channel)

On Exit: R0 preserved (or 0 if illegal channel number)
R1 = previous voice number (or 0 if not previously attached)

Attaching a new voice results in the previous voice being shut down and the new voice being reset. Different algorithms have different internal state representations so it is not possible to swap Voice Generator mid-sound.

A set of service calls provides efficient and centralised control of the the Level 1 Sound System. These control the Volume:

– scaled to the current system Volume setting

– pitch and overall tuning for voices that understand this parameter

– two forms of foreground control commands.

Sound_Volume – Set the overall loudness

Amplitudes are internally represented in a 7-bit logarithmic form with a change of 16 representing an effective doubling or halving of signal amplitude. Well behaved Voice modules should observe the volume setting and scale their waveforms to this maximum peak amplitude.

The default value is obtained from CMOS RAM to allow you to set a preferred loudness limit for the Sound system.

On Entry: R0 = sound volume (1 – 127) (0 to inspect last setting)

On exit: R0 = previous volume

– *Note*: this apparently trivial software interrupt does in fact perform large amounts of internal logarithm and lookup table calculation and scaling, and as such is an expensive operation. Therefore it is advisable to use this call to set the overall volume and then control the channel volumes directly.

Sound_SoundLog – Linear to audio logarithm

This software interrupt maps a 32-bit signed integer to an internal representation as a signed logarithmic byte value, scaled according to the current volume setting. (Table lookup for efficiency.)

On Entry: R0 = signed 32-bit signed number

On Exit: R0 = 8-bit signed scaled logarithm

Sound_LogScale – Internal audio logarithm scaling

This software interrupt maps an internal 8-bit sign and log representation number to one scaled to the current volume.

On Entry: R0b = signed audio logarithm

On Exit: R0 = 8-bit scaled audio logarithm

Sound_Pitch – Convert pitch to internal representation

This software interrupt maps a 15-bit pitch to an internal format pitch value (suitable for the standard voice phase accumulator oscillator).

On Entry: R0 = 15-bit pitch value: 127
bits 14 – 2 (3-bit octave number)
bits 11 – 0 (12-bit 1/4096 octave)

On Exit: R0 = 32-bit phase accumulator value

Sound_Tuning – Set the sound system tuning

This call sets the tuning parameter; this value is used to offset the pitch values used throughout the system.

On Entry: R0 = new tuning value (or 0 for no change)

n exit: R0 = previous tuning value

Sound_Control – Foreground (immediate) control of channel

This call allows real-time control of a specified Sound Channel. The parameters are immediately updated (they take effect on the next buffer fill entry) and may be used to provide real-time control of the amplitude, gating, pitch, and duration in a unified way for any voice type. Gating is used here as the term to describe new note on/off information as distinct from 'smooth' update control information; gate on corresponds to the distinct start of a note and gate off to stopping the note, whilst slurring corresponds to changing note parameters without restarting the note.

On Entry: R0 = channel number (1 – 8)
R1 = amplitude:
&FFF0 – &FFFF,0 for BBC emulation amplitude (0 to – 15)
&0001 – &000F ENVELOPE NOT EMULATED
&0100 – &01FF for full amplitude/gate control:
bit 7 is 0 for gate ON/OFF
1 for smooth update)
bits 6 – 0 are 7-bit audio log amp
R2 = pitch
&0000 – &00FF for BBC emulation pitch
&0100 – &7FFF for enhanced pitch control:
bits 14 – 12 = 3-bit octave
bits 11 – 0 = 12-bit fractional part of octave
(&4000 is nominally Middle C)
&8000 + n 'n' in range 0 – &7FFF as phase accumulator increment
R3 = duration
&0001 – &00FE for BBC emulation in 5 * centi-second periods
&00FF for BBC emulation 'infinite' time
> &00FF for 5 * n centi-seconds

On Exit: R0 – R3 preserved

Sound_ControlPacked – Foreground (immediate) control of channel

This call is identical to Sound_Control but the parameters are packed 16-bit at a time into low R0, high R0, low R1, high R1 respectively. This is then identical in format to the memory image of the OSWORD Sound call, and reflects the same packing of parameters into the Level 2 Scheduler format.

On Entry: R0 is AAAACCCC Amp/Channel
R1 is DDDDPPPP Duration/Pitch

On Exit: R0,R1 preserved

Sound_ReadControlBlock – Read channel control data

This call allows 32-bit data values to be read from the Sound Channel Control Block (SCCB) for the designated channel. The interpretation of many fields in the SCCB is not defined, and this call allows Voice Generators that require extended information (eg timbral control) over the general sound foreground interface, defined above, to communicate further data. Voice Generators that use extended SCCB parameters must document the extended interface, and it should be borne in mind that using extended parameters becomes non-general purpose across different voices. Use with care.

On Entry: R0 = channel
R1 = offset to read from

On Exit: R0 preserved (or 0 if fail, invalid read offset)
R1 preserved
R2 = 32-bit word read (if R0 non-zero)

Sound_WriteControlBlock – Write channel control data

This call allows 32-bit data values to be written to the Sound Channel Control Block (SCCB) for the designated channel. See previous SWI description.

On Entry: R0 = channel
R1 = offset to read from
R2 = 32-bit word to write

On Exit: R0 preserved (or 0 if fail, invalid write offset)
R1 preserved
R2 = previous 32-bit word (if R0 non-zero)

SoundChannels Level 1 control block

SoundChannels registers a Level 1 control block with Level 0 which has the first four entries defined; in addition to the mandatory Fill and Fixup entries, two pointers are provided to lookup tables which are internally scaled according to the current volume.

The SoundLevel1LogTable pointer is to the base of an 8 kbyte table arranged to map signed linear numbers directly to volume-scaled 8-bit audio logarithms suitable for VIDC buffer filling. (See the section **Sound voice generators** for coding implementation details.)

The SoundLevel1AmpTable pointer is to the base of a 256-byte table which maps a VIDC-format sign and magnitude audio logarithm amplitude from maximum range down to a value scaled to the volume setting.

```
; Level 1 data structure
SoundLevel1FillPtr      =  0 ; pointer to Level 1 code
SoundLevel1FixupPtr     =  4 ; pointer to overrun fixup code
SoundLevel1LogTable     =  8 ; pointer to Linear-to-Log table
SoundLevel1AmpTable     = 12 ; pointer to Log-scale table
```

Sound channel control block (SCCB)

Each sound channel has a control block which contains a common set of basic parameters and flags that Voice Generators use with an extension area. Voices interpret these in specialised ways. There is also a Flags field which indicates the state of the voice and may be used for allocating voices in a polyphonic manner or checking for Overrun errors (voice unable to complete processing in real time – an external scheduler could inspect overrun fields and de-allocate voices which continually fail to fill in time).

```
; Sound Channel Control Block : SCCB
; 8 initial words (normally for LDMIA R9,{R0-R7} entry)
; size constrained to exactly 256 bytes
SoundChannelAmpGateB     =  0  ; gate + 7-bit log amp.
SoundChannelVoiceIndexB =  1  ; index to voice table
SoundChannelInstanceB    =  2  ; Voice instance no.
SoundChannelFlagsB       =  3  ; control/status bit flags
SoundChannelPitch        =  4  ; phase acc pitch oscillator
SoundChannelTimbre       =  8  ;
SoundChannelDuration     = 12  ; no. of buffer fills (counter)
SoundChannelReserved4    = 16  ; (normally working R4)
SoundChannelReserved5    = 20  ; (normally working R5)
```

```
SoundChannelReserved6    = 24  ; (normally working R6)
SoundChannelReserved7    = 28  ; (normally working R7)
SoundChannelReserved8    = 32  ; (normally working R8)
; ACORN reserved area follows
SoundChannelExtension    = 64  ; START HERE -> 256 BYTES
SoundChannelCBSize       = 256 ; total size of SCCB
```

See **Sound voice generators** in the next section for details of usage.

SOUNDSCHEDULER (LEVEL 2) – SOUND SYSTEM SCHEDULER

All the tempo- or time-related sound system facilities are supported by the Level 2 module which is responsible for ordering and synchronising events scheduled for future activation and depending on the defined tempo activating the Level 1 interface in real-time. Note on/note off events are those that are normally processed, but the system is made more powerful by the ability to issue sound system SWI calls (or even general SWI calls to enable synchronised graphics, or external musical instrument control) subject to certain constraints.

Level 2 * commands

Commands are provided as follows:

*TEMPO

The *Tempo command sets the sound scheduler tempo.

Syntax: *Tempo <n> (0 – &FFFF, &1000 Default)

The default tempo is &1000, which corresponds to one microbeat per centi-second; doubling or halving the value proportionally affects the rate at which scheduled events are played back.

***QSOUND**

*QSound queues a sound after the specified number of beats.

Syntax: *QSound <chan> <amp> <pitch> <duration> <nBeats>

This OSCLI command is effectively equivalent to the five parameter BASIC SOUND command; all parameters must be unsigned numbers.

Level 2 SWI calls

A series of software interrupt service entries are provided which allow tempo-related and synchronised music to be produced.

Sound_QInit – Flush and initialise the event queue

This call flushes out all events currently scheduled and re-initialises the event queue data structures. The Tempo and Beat variables are reset to their default values.

On entry: –

On Exit: R0 = 0, indicating success

Sound_QSchedule – Schedule a sound event

This call attempts to schedule an 'event' on the Sound Event Queue for activation in the specified number of tempo-controlled beat period ticks. The schedule time is treated as beat counts since the last beginning of a bar; after initialisation the beat counter is reset and disabled, so all scheduling will occur in tempo intervals from time 'now'. If a second (or further) event is to be synchronised with the last, then a schedule time of –1 forces the new event to be queued for activation concurrently with the last one.

The event is typically a Sound_ControlPacked type call, although any other sound (or for that matter, ANY general) SWI number plus the register contents for R0 and R1 (ONLY; registers 2 – 7 are cleared when the SWI is activated) may be used. There are, of course, practical implications: most of the Sound SWI calls are defined to use registers R0 and R1 for basic operation (with the other registers ignored or treated as 'don't change parameters if 0'); any return parameters are ignored. Register contents that are pointers may well become catastrophic if the memory contents are not guaranteed until the schedule period is complete.

On Entry: R0 = the schedule period (from last beat 0) or –1 to synchronise with the last scheduled event
R1 = 0 to cause a Sound_ControlPacked call or the SWI code to schedule (of the form &xF000000 + SWI no.)
R2,R3 are the SWI parameters for R0,R1

On exit: R0 = 0 for successfully Queued
R0 < 0 for failure (Queue full)

Sound_QRemove – Reserved Level 2 call

This call is internally reserved for internal scheduler operation.

Only for custom Sound Level0 Handlers.

On Entry: R0 = incremental slot advance period

On Exit: R0 = incremental slot advance period remaining
R1 – R3 scheduled data
or
R0 < 0 for no events to remove (and slots advanced)
R1 – R3 indeterminate

Sound_QFree – Check free slots

This call returns the most pessimistic number of slots guaranteed free at the current instant (takes into account worst case data structure overheads). The caller is welcome to exceed the guaranteed free slot count (and cause this call to return negative values!) but QSchedule status must be carefully monitored to observe when overflow occurs.

On entry: –

On Exit: R0 = no. of guaranteed slots available
R0 < 0 indicates over worst case limit but OK

Sound_QDispatch – Reserved Level 2 call

This call is used internally to advance scheduler time by the next tempo-related period, and to dispatch all events scheduled for this period.

Only for custom Sound Level0 Handlers.

On entry: –

On exit: Tempo advanced and any pending SWIs activated

Sound_QTempo – Set the sound system tempo

This call sets the tempo parameter which is used by the Level2 Scheduler. The parameter should be treated as a hexadecimal fractional number, where the three least-significant digits are the fractional part. A value of &1000 corresponds to a tempo of one tempo beat per centi-second. Doubling the value causes the tempo to double (2 tempo beats per centi-second), while halving the value halves the tempo (to half a beat per centi-second).

On Entry: R0 = new tempo (or 0 for no change)

On Exit: R0 = previous tempo value

Sound_QBeat – Set/read the tempo beat counter

This call sets the tempo beat counter, or allows the beat counter to be read. The beat counter, when enabled, simply increments from 0 up to the programmed count number (N–1), then resets and begins counting again. The beat counter enables synchronisation of musical events as they are scheduled, and the effect of the counter reaching the programmed counter limit causes it to reset. Sound event scheduling is always relative to the last tempo beat reset.

After the initialisation counter is disabled, the call will always return beat 0 and all scheduling will always be relative to the present time. When the counter is programmed with a positive value the beat count will increment and wrap (with the option of causing an event each time the counter wraps). The beat counter may be disabled by specifying a negative count (less than –1 in fact).

On Entry:	R0 = 0 to return current beat number R0 = –1 to return the current beat counter value R0 < 0 to clear and disable the beat counter (to 0) RO = +N for beat set count of N (counts 0 to N–1)
On Exit:	R0 = current beat number if input parameter was 0, otherwise the previous beat Count value

The event queue

The central time-ordered Event Queue manager is provided in order to allow the scheduling and synchronisation of sounds on all sound channels. Schedule times are always relative to the beat counter zero-crossing point. Although principally designed for queuing sound commands to internal channels, a powerful software interrupt scheduling scheme is implemented to allow easy extension of synchronised activity to, for example, an external instrument interface (such as a Musical Instrument Digital Interface (MIDI) expansion podule), or a screen-based music editor with real-time score replay.

Events are de-queued every sound system scheduler period, and the voice algorithm and all parameter change events are first processed and used to update the Voice Control Block before the buffer fill code is activated.

The event queues are implemented as linked list structures with fixed sized records that are recycled after use via a free list stack. The number of free slots varies according not only to how many events are presently queued but also to how the events are 'clustered'. The queue is implemented as a circular list of bucket chains, with buckets accumulating all events scheduled to occur at a given microbeat.

Event Queue Record, in a scheduler bucket (R0 schedule period)

link	pointer to next record
SWI	copy of queued R1
SWI R0 param	copy of queued R2
SWI R1 param	copy of queued R3

Event dispatcher

Every centi-second the 'scheduler time' is advanced according to the current tempo value, and any events that fall in the advance period are activated in strict queuing order. The software interrupt calls are issued using auto error-handling, as they are unable to report errors explicitly; return parameters are discarded. Scheduled software interrupts are issued with R0 and R1 set to copies of the Sound_Schedule R2 and R3 parameters, and registers R2 to R7 are cleared to zero prior to the SWI call. Most sound system interface SWI's can be scheduled in this way, but care should be taken to check whether pointers are used; these are fatal if the data to which they point is not guaranteed to be preserved until the SWI is actually activated. This becomes even more serious when external SWI calls to other modules are issued.

Sound voice generators

A Voice Generator is an assembler-coded algorithm for synthesizing one or more streams (if the multiple instantiations of the voice are permitted) of bytes on demand to the SoundChannel handler once they have been both installed and attached to an active sound channel.

The voice generator must provide a header block containing a fixed number of entry points used both for attaching and detaching the voice under supervisor mode control, plus a set of real-time buffer filling entries which are entered in IRQ mode to perform the actual synthesis and buffer filling operations.

Associated with the voice header is a name string that is used as the textual reference in the Level 1 voice table. The name should be reasonably descriptive and also concise.

The speed and efficiency of the generator algorithms is paramount, and requires careful attention to coding. ROM voice generators in fact copy an image of the synthesis kernel into system heap RAM, in order that the code executes in fast sequential memory modes.

Voice libraries are efficient for sharing common code and channel instance data areas, and would normally be built as Relocatable Modules which install sets of voices preferably with some form of library name prefix.

Voice generator header block

The first eight words of the voice generator header must conform to the format described below. The first four entries are the real-time IRQ mode buffer fill entries, the next three are SWI mode interface procedure entries, and the eighth is a relative offset of the voice name string to the voice base address.

Voices are installed by registering a pointer to the header block using the SWI Sound_Install call (see Level 1) and must be removed using SWI Sound_Remove if the voice is to be unloaded (normally a Voice Generator is a Relocatable Module and any attempts to clear or tidy the RMA, or kill off the voice module must register removal and installation with Level 1).

```
; Level 2 (VOICE) data structure
; SVCB Sound Voice Control Block
 .VoiceBase
 .VoiceFillEntry           B FillCode
 .SoundVoiceUpdate         B UpdateCode
 .SoundVoiceGateOn         B StartCode
 .SoundVoiceGateOff        B ReleaseCode
 .SoundVoiceInstantiate    B Instantiate
 .SoundVoiceFree           B DeInstantiate
 .SoundVoiceInstall        LDMFD R13!,{pc} ; not supported
 .SoundVoiceTitle          EQUD VoiceName - VoiceBase
```

Voice generator buffer filling

The following register allocation is specified on entry to any one of the first four (IRQ mode) buffer fill code entry points.

Register	Function
R6	(negative if Level1 configure changed)
R7	Channel number
R8	Sample Period in uS
R9	Pointer to Voice Code Base
R10	Buffer End Pointer
R11	Buffer Increment
R12	Buffer (Write) Pointer
R13	Stack (Return address on top of stack)
R14	** DO NOT USE **

Return address on top of stack (LDMFD R13!,{pc})

– *Note*: IRQ's are enabled on entry, so R14 is untrustworthy

A voice generator is a re-entrant code section which is called by the channel handler with all parameters (or pointers to parameters) in registers with a request for a given number of 8-bit data samples. In order to optimise the buffer filling speed for each channel, the voice generator is presented with a pointer to the virtual memory address to start filling from (which includes the buffer interleave offset set up by the channel handler). In addition, the buffer increment and the buffer address limit values are passed in, these being the only other parameters required by the voice generator to generate the stream of data samples for output.

Because code execution speed is of key importance in real-time buffer filling, the sound voice generator code would not run fast enough in ROM; a ROM template or user code template has to be copied into the stream code block (subject to a large enough allocated code segment space).

The code runs in IRQ mode, but IRQs are enabled to ensure that IRQ latency is not made unreasonable for other system devices. This means that R14 cannot be used as a subroutine link register, since it will be corrupted by an interrupt. Normal voice code should be carefully written so that the IRQ link register is not used and to ensure sufficient system IRQ stack depth is maintained for other system IRQ handling.

R9 points to the Sound Channel Control Block (SCCB) for the appropriate channel as described in the SoundChannels specification. Parameters are exchanged between Level 1 and the Voice Generator, and the parameter usage depends on which entry point has been called.

On fill completion, a flags byte must be returned to Level 1 in R0 to indicate filling status. The flags byte is used by the system to prioritise the real-time fill requests to the attached Voice Generator. Return to level 1 is then performed by LDMFD R13!,{PC}.

```
; Channel Flags Byte: (return through R0)
;
;   7                             0
; +---+---+---+---+---+---+---+---+
; | Q | K | I | F | A | V | F2| F1|
; +---+---+---+---+---+---+---+---+
; Z - Quiet (inactive)
; K - Kill Pending
; I - Initialise Pending
; F - Fill Pending
; A - Active Flag
; V - oVerrun Flag
; F2,F1 - 2-bit Flush Pending counter

SoundChannelGateOff      = &80 ; carefully priority-encoded
SoundChannelGateOn       = &40
SoundChannelUpdate       = &20
SoundChannelReserved     = &10
SoundChannelActive       = &08 ; normal continuation fill in progress
SoundChannelOverrun      = &04
SoundChannelFlush2       = &02 ; 2-bit flush counter
SoundChannelFlush1       = &01
```

GateOn entry

The GateOn entry is used whenever a new sound command is issued on the specified channel, and the SCCB parameters for amplitude, pitch and duration are updated. Normally any previous synthesis on this channel would be aborted and the algorithm restarted with the new parameters.

On exit the duration parameter would normally be updated and a SoundChannelActive flags byte would be returned ready for continued filling next buffer fill period.

Fill entry

The Fill entry is the normal entry used when Level 1 requests the next sample buffer fill; no new sound commands are pending on the specified channel, and the SCCB parameters for amplitude, pitch and duration would normally be updated at the completion of each fill.

On exit, the duration parameter would normally be updated and a SoundChannelActive flags byte would be returned. However, if the duration period has expired and the waveform has decayed to zero amplitude, then it is usual to return with a SoundChannelFlush2 flags byte so that Level 1 will automatically flush out the next two DMA buffers for this channel before becoming dormant.

Update entry

The Update entry is used whenever a sound command is issued for smooth update of a parameter (or parameters) rather than a new GateOn command (using extended amplitudes &180 to &1FF in the SOUND command for example). The SCCB parameters for amplitude, pitch and duration would normally be reloaded by the Voice Generator, but the rest of the local state for this channel instance would normally be preserved.

Exit flags as for Fill Entry.

GateOff entry

The GateOff entry is used to force the Voice Generator to close down the the sound on this channel instance. Simple voices may stop abruptly (but this is liable to cause an audible 'click'), whilst more refined algorithms might enter a specific note release phase (which might require buffers filling for a number of buffer periods).

On exit, the duration parameter would normally only return a SoundChannelFlush2 flags byte if the sound is simply to be truncated, but could return SoundChannelActive as the release phase is entered. Once this channel has received a GateOff call it might well receive a new GateOn call directly.

Voice instantiation

Two entry points are provided to inform the voice generator that a request has been issued to attach or detach a sound channel to it. Voice Generators usually require some private workspace per channel; very complex algorithms or, for example, a voice generator which plays back data samples from disc, may only be able to support one channel instance. Other generators may be instantiated up to eight times directly. The Instantiate entry gives the voice generator the option of accepting or rejecting a request to attach a physical channel to an instance of the algorithm. The Free entry is simply called to inform the voice generator that the channel which has been attached is now to be reattached elsewhere, and the instance is to be killed off. The free call MUST release the channel successfully.

The entries are only made in supervisor mode, and any registers to be used must be preserved explicitly using the SVC stack normally.

Register	Function
R0	physical Channel number –1 (0 to 7)
R14	usable

Return address on top of stack (LDMFD R13!,{pc})

Instantiate Entry

The instantiate entry is called with R0 containing the number of the physical channel (in fact channel –1) to which it is requested to attach an instance of this voice. If a new instance cannot be created for this channel, then the channel number in R0 should be changed. The test for successfully instantiating the channel is equivalence of R0 before and after the instantiate call.

Free entry

The instantiate entry is called with R0 containing the number of the physical channel (–1) to be freed, and this call does not have the option of failing (all registers, including R0 should be preserved).

Install Entry

There is one other entry point defined, originally to be called when a voice was installed. However, Relocatable Modules, which are envisaged as the normal way in which user-developed voices are to be implemented, offer exactly this service in the form of the Initialistion Entry point; thus this field is not to be supported.

Buffer filling algorithms

Each voice generator has Virtual Memory addressing access to the entire output buffer (and so has the potential to corrupt any of the sound data samples already interleaved throughout the buffer). The sound stream handler sets up three registers (R12,11,10) to give the start address, the buffer write interleave increment and the end address for correct filling. In the general case, each voice generator should include something like the following in-line code:

```
.loop
        ...
        ...                    ; e.g. form byte in rl
        STRB  R1,[R12],R11     ; store, and bump ptr
        CMPS  R12,R10          ; check for end
        BLT   loop             ; and loop until fill complete
```

To avoid the loop overheads every sample, the following points should be borne in mind:

The total DMA buffer length is always a multiple of 16, so two-byte store operations may always be performed in the loop code. The system Level 0 Handler in fact guarantees that buffers will always be filled to a multiple of four bytes, and that the system default buffer byte count of 208, the in-line loop code, may be extended considerably.

The increment value (in r11) will be either 1, 2, 4, or 8. A quick test to see if the value is unity immediately ascertains whether this is the only voice active, implying that a sequential store is required. If this is the case, then WORD stores may be used, dramatically improving the buffer filling time. (This is possible because hardware buffers are always guaranteed to be a multiple of 16 bytes long, always word-aligned.)

Voice generator code fragments

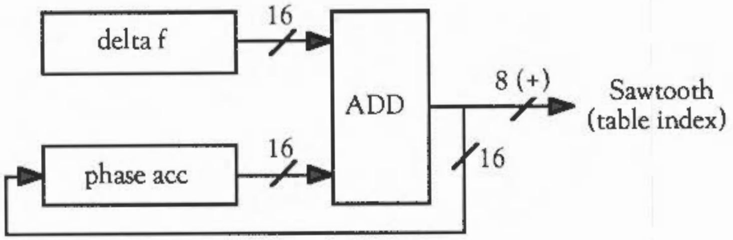
The fundamental operations performed by nearly all voice generators involve Oscillators, Table lookup and Amplitude modulation. In addition, some algorithms (plucked string and drum in particular) require random bit generators. Simple in-line code fragments are briefly outlined for each of these.

– *Note*: in all cases the aim is to produce the most efficient, and wherever possible highly sequential, ARM machine code; in most algorithms the aim must be to get as many working variables into registers as possible, and then adapt the synthesis algorithms wherever possible to use the high-speed barrel shifter to effect.

Oscillator code

The accumulator-divider is the most useful type of oscillator for most voices. Basically, a frequency increment is added to a phase accumulator register and the high-order bits of the resulting phase (modulo wavetable length) provide the index to a wavetable. The frequency increment is linearly related to the oscillator frequency. The result is that the oscillator may be given the correct frequency, or phase modulated, directly by adding in the modulating component. Sixteen-bit registers provide good audible frequency resolution, and are used in many digital hardware synthesizer products. The 32-bit register width of the ARM is ideally split 16/16 bits for phase/increment.

Schematically:



Coding:

Register field assignment: Rp

31 16	15 0
PHASE ACCUMULATOR	INCREMENT

```
ADD    Rp,Rp,Rp,LSL #16  ; phase accumulate
```

In many cases it is desirable to update wavetable pointers or parameters at or close to zero-crossing points: if wavetables are arranged with zero-crossing aligned to the start and end of the wavetable then it is a simple matter to add a single S (sequential) cycle.

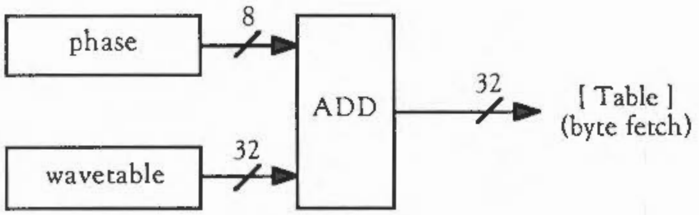
Coding:

```
ADDS   Rp,Rp,Rp,LSL #16  ; phase accumulate
BCS    Update  ; only take branch if past zero crossing
```

Waveform table access code

Normally fixed-length (256-byte or a larger power of two) wavetables are used by most voice generator modules. Given a wavetable base pointer and the way the oscillator phase increment is calculated, it is possible in one single pre-indexed load instruction to access the wavetable sample byte:

Schematically: (for a 256-byte table)



Coding:

Given the Phase index calculated in the most significant 8 bits and Rt as the Table base pointer:

```
LDRB Rn,[Rt,Rp,LSR #24]
```

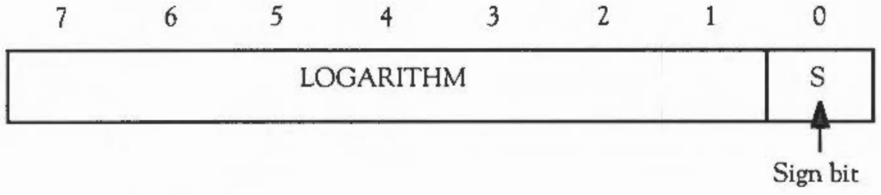
where Rn is the destination register.

Amplitude modulation code

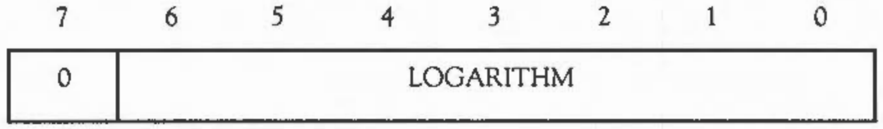
Amplitude modulation involves the multiplication of the current sample value magnitude by the amplitude envelope component, with the internal logarithmic representation used by the output DAC. This process simplifies to addition of log values and range checking of the result (for result under/overflow). In fact, the numeric values are treated as fractional quantities (in the range 0 to 1) and underflow is the only condition of concern. For example, when values become too small to be correctly represented (-ve log values), the result must be rounded to 0. The amplitude value must, in fact, be pre-processed to give an effective division (peakAmp minus amplitude, where peakAmp is 127 for maximum, rounded up to zero if negative).

The data format for the VIDC sound data byte stream encodes the sign bit as the least significant bit and the 7-bit logarithmic magnitude value as the top seven bits (bit 7 corresponds to the most significant amplitude bit).

VIDC 8-bit sample format



Amplitude Byte Data Format:



Coding:

Ra contains amplitude in range 0 to 127
Rs contains sample data in range –127 to +127 [sign bit LSB]

```
SUBS  Rs,Rs,Ra,LSL #1
MOVMI Rs,#0
```

– *Note*: the SoundChannels Level 1 module provides a pointer to a 256-byte volume scaled lookup table and an example of usage is given in the worked example at the end of this section.

Linear-to-Logarithmic conversion

Algorithms which work with linear integer arithmetic may use the Level 1 SoundChannels module linear-log table directly to fill buffers efficiently. The table is 8 kbyte in length, to allow the full dynamic range of the VIDC sound DAC to be utilised, and the format is chosen to allow direct indexing using barrel-shifted 32-bit integer values. The lookup characteristic is varied according to the current volume setting.

Coding:

```
   ; to access the lookup table pointer during initialisation:
        MOV     R0,#0
        MOV     R1,#0
        MOV     R2,#0
        MOV     R3,#0      ; get Level 1 base
        MOV     R4,#0
        SWI     "Sound_Configure"
        LDR     R8,[R3,#8] ; lin-to-log pointer

  ; in line buffer filling code:
  ; linear 32-bit value in R0
        LDRB    R0,[R8,R0,LSR #19] ; lin -> log
        STRB    R0,[R12],R11       ; output
```

Random bit generator code

An efficient multi-tap 32-bit shift register pseudo-random bit generator (with a sequence length of 4294967295) can be implemented to run fast using two internal registers. One register would be loaded up with the current seed value, and the second with an XOR bit mask constant (&1D872B41); random carry bit setting by the simple code fragment outlined below will allow conditional execution on carry set (or cleared) directly.

Coding:

```
MOVS  R8,R8,LSL #1 ; set random carry
EORCS R8,R8,R9
xxxCC  ; do this..
yyyCS  ; ..or alternately this
```

Example voice

```
REM -> WaveVoice
:
DIM WaveTable% 255
DIM Code%     4095
:
FOR s%=0 TO 255
  SYS "Sound_SoundLog",&7FFFFFFF*SIN(2*PI*s%/256) TO WaveTable%?s%
NEXT s% : REM build samples
:
FOR a%=WaveTable% TO WaveTable%+255
  SYS "Sound_LogScale",?a% TO ?a%
NEXT a% : REM scale to current volume
:
FOR C=0 TO 2 STEP 2
P%=Code%
[ OPT C
;*************************************
;*   VOICE CO-ROUTINE CODE SEGMENT    *
;*************************************
; On installation, point Level1 voice
; pointers to this voice control block
; (return address always on top of stack)
.VoiceBase
   B       Fill
   B       Fill                 ; update entry
   B       GateOn
   B       GateOff
   B       Instance             ; Instantiate entry
   LDMFD   R13!,{PC}            ; Free entry
   LDMFD   R13!,{PC}            ; Initialise
   EQUD    VoiceName - VoiceBase
;
.VoiceName EQUS "WaveVoice"
           EQUB 0
   ALIGN
;*************************************
```

```
.LogAmpPtr EQUD 0
.WaveBase  EQUD WaveTable%
;**************************************
.Instance ; any instance must use LogAmp table
   STMFD   R13!,{R0-R4}
   MOV     R0,#0
   MOV     R1,#0
   MOV     R2,#0
   MOV     R3,#0
   MOV     R4,#0
   SWI     "Sound_Configure"
   LDR     R0,[R3,#12]        ; Level1 ScaleTable Offset
   STR     R0,LogAmpPtr
   LDMFD   R13!,{R0-R4,PC}    ;restore and return
;**************************************
;*  VOICE BUFFER FILL ROUTINES        *
;**************************************
; on entry:
;   r0-r8 available
;   r9  is SoundChannelControlBlock pointer
;   r10 DMA buffer limit (+1)
;   r11 DMA buffer interleave increment
;   r12 DMA buffer base pointer
;   r13 Sound System Stack with return address and flags
;     on top (must LDMFD R13!,{...,pc}
; NO r14 - IRQs are enabled and r14 is not usable
.GateOn
   LDR     R0,WaveBase        ; wavetable base
   STR     R0,[R9,#16]        ; set up as working register 5
   LDR     R0,LogAmpPtr       ; volume scaled log amp table
   STR     R0,[R9,#20]        ; set up as working register 6
;**************************************
.Fill
   LDMIA   R9,{R1-R5}         ; pick up working registers
   AND     R1,R1,#&7F
; R1 is amp (0-127)       R2 is pitch phase acc
; R3 is timbre phase acc  R4 is duration
; R5 is wavetable base    R6 is amp table base
```

```
   LDRB    R1,[R6,R1,LSL #1] ; get volume scaled amp
   MOV     R1,R1,LSR #1      ; R1 is volume scaled amp (0-127)
   RSB     R1,R1,#127         ; make attenuation factor
.FillLoop
   ADD     R2,R2,R2,LSL #16   ; advance waveform phase
   LDRB    R0,[R5,R2,LSR #24] ; get wave sample
   SUBS    R0,R0,R1,LSL #1    ; scale amplitude
   MOVMI   R0,#0              ; and correct underflow
   STRB    R0,[R12],R11       ; generate output sample
   ADD     R2,R2,R2,LSL #16   ; repeated in line four times...
   LDRB    R0,[R5,R2,LSR #24]
   SUBS    R0,R0,R1,LSL #1
   MOVMI   R0,#0
   STRB    R0,[R12],R11
   ADD     R2,R2,R2,LSL #16
   LDRB    R0,[R5,R2,LSR #24]
   SUBS    R0,R0,R1,LSL #1
   MOVMI   R0,#0
   STRB    R0,[R12],R11
   ADD     R2,R2,R2,LSL #16
   LDRB    R0,[R5,R2,LSR #24]
   SUBS    R0,R0,R1,LSL #1
   MOVMI   R0,#0
   STRB    R0,[R12],R11
   CMP     R12,R10            ; check for end of buffer fill
   BLT     FillLoop           ; loop if not
; check for end of note
   SUBS    R4,R4,#1           ; decrement centisec count
   STMIB   R9,{R2-R5}
   MOVPL   R0,#%00001000      ; voice active
   MOVMI   R0,#%00000010      ; force flush
   LDMFD   R13!,{PC}          ; return to level 1
;***************************************
.GateOff
   MOV     R0,#0
.FlushLoop
   STRB    R0,[R12],R11
   STRB    R0,[R12],R11
   STRB    R0,[R12],R11
   STRB    R0,[R12],R11
   CMP     R12,R10
   BLT     FlushLoop
; CAUSE level 1 TO FLUSH once more
   MOV     R0,#%00000001       ; flush one more buffer
   LDMFD   R13!,{PC}           ; return to level 1
]
NEXT C
:
DIM OldVoice%(8)
SYS "Sound_InstallVoice",VoiceBase,0 TO a%,Voice%
FOR v%=1 TO 8
  SYS "Sound_AttachVoice",v%,0 TO z%,OldVoice%(v%)
  VOICE v%,"WaveVoice"
NEXT
:
ON ERROR PROCRestoreSound : END
:
VOICES 8
*voices
SOUND 1,&17F,53,10 :REM activate channel 1!
PRINT''"any key to make a noise, <ESCAPE> to finish"
:
C%=1
REPEAT
  K%=INKEY(1)
  IF K%>0 THEN
    SOUND C%,&17F,K%,100
    C%+=1 : IF C%>8 THEN C%=1
  ENDIF
UNTIL 0
:
DEF PROCRestoreSound
  ON ERROR OFF
  REPORT:PRINT ERL
  SYS "Sound_RemoveVoice",0,Voice%
  FOR v%=1 TO 8
```

```
    SYS "Sound_AttachVoice",v%,OldVoice%(v%)
  NEXT
  VOICES 1
  *voices
  PRINT''
ENDPROC
```

THE DEBUGGER

This chapter describes the facilities provided by the machine code debugger module. Most of these are accessed through * commands. There is one SWI provided by the debugger, though.

The debugger allows breakpoints to be set so that a piece of code will stop when it reaches a particular instruction. Other commands may then be called to interrogate and even reset the values contained at particular addresses in memory and to list the contents of the registers. Then execution of the code may be continued from that point.

THE DEBUGGER * COMMANDS

Command	Description
*BREAKCLR	Remove breakpoint
*BREAKLIST	List currently set breakpoints
*BREAKSET	Set a breakpoint at a given address
*CONTINUE	Start execution from a breakpoint saved state
*DEBUG	Enter the debugger
*INITSTORE	Fill memory with given data
*MEMORY	Display memory between two addresses/register
*MEMORYA	Display and alter memory
*MEMORYI	Disassemble ARM instructions
*QUIT	Perform a SWI Exit
*SHOWREGS	Display registers caught by traps

When an address is required, it should be given in hex, without a preceding &.

*BREAKCLR

Syntax: *BREAKCLR [<addr>]

*BREAKCLR removes the breakpoint at the specified address, putting the original contents back into that location. If no address is given then all breakpoints are removed.

Note that debugger breakpoints are separate from those caused by OS_BreakPt: you can't continue from the latter using *CONTINUE.

*BREAKLIST

Syntax: *BREAKLIST

*BREAKLIST lists all the breakpoints currently set, in the form:

```
Address     Old Data
00008704    EF000141
```

There may be up to 16 breakpoints set at once.

*BREAKSET

Syntax: *BREAKSET <addr>

*BREAKSET sets a breakpoint at the address given so that when the code is executed and the instruction at that address is reached, execution will be halted.

The previous contents of the breakpoint address are replaced with a branch into the debugger code. This means that breakpoints may only be set in RAM. If you try to set a breakpoint in ROM, the error `Bad breakpoint address` will be given.

When a breakpoint instruction is reached, the debugger is entered, with the prompt `DEBUG*`, from which you can type any * command. An automatic register dump is also displayed.

*CONTINUE

Syntax: *CONTINUE

*CONTINUE starts execution from the breakpoint saved state. If there is a breakpoint at the continuation position, then this prompt is given:

```
Continue from breakpoint set at &0008704
Execute out of line? [Y/<anything>]
```

Reply 'Y' if it is permissible to execute the instruction at a different address (ie it does not refer to the PC). If the instruction that was replaced by the breakpoint contains a PC-relative reference (eg LDR R0,label or an ADR directive), then you should reset the break point before continuing. This causes the instruction to be executed in-line, otherwise the wrong address is referenced.

***DEBUG**

Syntax: *DEBUG

This command enters the debugger. It will be expanded from the present form of allowing you to type * commands only in a future release.

***INITSTORE**

Syntax: *INITSTORE [<data>]

*INITSTORE fills user memory with the specified data or the value &E1000090 (which is an illegal instruction) if no parameter is given. If you give this command from within an application (eg BASIC), the machine will crash, and will have to be reset.

***MEMORY**

Syntax: *MEMORY [B] <addr1/reg1> [[+]<addr2/reg2>]

*MEMORY displays the values in the memory in ARM words from the address given either explicitly or contained in a register to the next address given in <addr2/reg2>. If the second address begins with a '+', this address is added to the first. Otherwise it is taken to be an absolute value. If no second address is given 256 bytes are displayed.

If the optional B is given after the command but before the start address, the display is byte-oriented, with 16 bytes per line. If it is omitted, the display is grouped as bytes, with four words to a line. For example:

```
*MEMORY 1000 +200
```

*MEMORYA

Syntax: *MEMORYA [B] <addr/reg1> [<data>/<reg2>]

*MEMORYA displays and alters the memory in bytes, if the optional B is given, or in words otherwise. It starts at the address given absolutely or within a register. If no further parameters are given, interactive mode is entered where the following may be typed:

↵	to go to 'next' location
–	to step backwards in memory
+	to step forwards in memory
<hex digits>	to alter a location and proceed,
<anything else>	to exit

At each line, the following is printed:

```
+ 000087A0 : cccc : xxxxxxxx : opcode
  Enter new value :
```

where the '+' is the direction in which ↵ steps (it is '–' for backwards). Next is the address of the word/byte being altered, then the four characters in that word, then the current value of the word, and finally the instruction at that address.

In byte mode, it looks like this:

```
+ 000087A1 : c : xx
```

Alternatively you can give the new data value on the line after the address, eg:

```
*MEMORYA 87A0 123456578
```

*MEMORYI

Syntax: *MEMORYI <addr1/reg1> [[+]<addr2/reg2>]

*MEMORYI disassembles ARM instructions starting at the location given in <addr1/reg1> until it reaches the address in the second parameter if given. If the second parameter is missing, it continues disassembling for 25 instructions.

*QUIT

Syntax: *QUIT

*QUIT exits the debugger. It returns to the last routine which claimed the exit handler by performing a SWI Exit.

*SHOWREGS

Syntax: *SHOWREGS

*SHOWREGS displays the registers caught on one of the five following traps:

- unknown instruction
- address exception
- data abort
- address abort
- breakpoint.

It also prints the address in memory where the registers are stored, so you can alter them (for example after a breakpoint) by using *MEMORYA on these locations, before using *CONTINUE.

Debugger_Disassemble &40380

This is the only SWI provided by the debugger.

On entry: R0 = instruction to disassemble

On exit: R1 = address of buffer containing text
R2 = length of disassembled line

R0 contains the 32-bit instruction to disassemble. On exit, R1 points to a buffer which contains a zero terminated string. This string consists of the instruction mnemonic, and any operands, in the format used by the *MEMORYI instruction. The length in R2 includes the zero-byte.

THE FLOATING POINT EMULATOR

The Acorn RISC machine has a general co-processor interface. The first co-processor envisaged is one which performs floating point calculations to the IEEE standard. To ensure compatability with future versions of the RISC machine which use this co-processor, the machine contains a floating point emulator module which provides all the functionality of a hardware processor. The instructions it provides may be incorporated into any assembler text, provided they are called from user mode. However, these instructions are not supported by the BASIC assembler and are given here for information only.

INTRODUCTION

PROGRAMMER'S MODEL

The ARM IEEE floating point system has eight 'high precision' floating point registers, F0 to F7. The format, in which numbers are stored in these registers, is not specified. Floating point formats only become visible when a number is transferred to memory, using one of the precisions described below.

There is also a floating point status register. This is used to hold flags which indicate various error conditions, such as overflow and division by zero. Each flag has a corresponding mask, which can be used to enable or disable a 'trap' associated with the error condition.

Precision

All basic floating point instructions operate as though the result were computed to infinite precision and then rounded to the length and in the way, specified by the instruction. The rounding is selectable from:

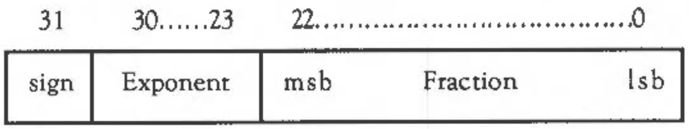
- Round to nearest
- Round to +infinity (P)
- Round to –infinity (M)
- Round to zero (Z).

The default is 'round to nearest'. If any of the others are required they must be given in the instruction.

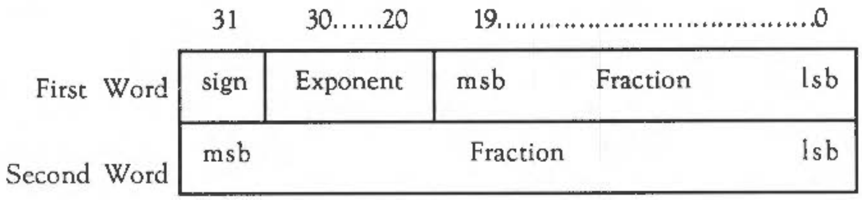
The working precision of the system is 80 bits, comprising a 64 bit mantissa, a 15 bit exponent and a sign bit.

Like the ARM instructions, the floating point data processing operations refer to registers rather than memory locations. Values may be stored into ARM memory in one of four formats:

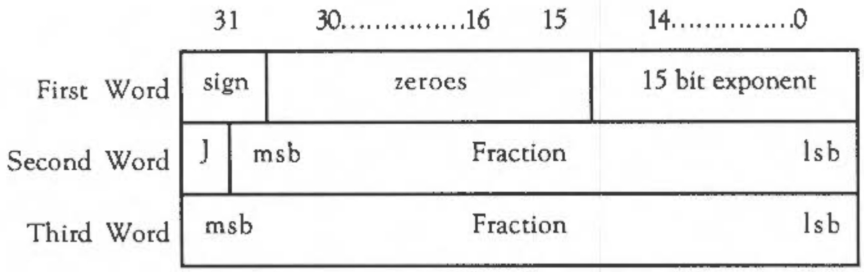
– IEEE Single Precision (S)



– IEEE Double Precision (D)



– Double Extended Precision (E)



J is one bit to the left of the binary point

Storing a floating point register in 'E' format is guaranteed to maintain precision when loaded back into the floating point system in this format.

– Packed Decimal (P)

31..0

First Word	sign	e3	e2	e1	e0	d18	d17	d16
Second Word	d15	d14	d13	d12	d11	d10	d9	d8
Third Word	d7	d6	d5	d4	d3	d2	d1	d0

Value is :

+/– d * 10 ^ (+/– e)

d18 or e3 is the most significant digit. Sign contains both the number's sign (top bit) and the exponent's sign (next bit). The other two bits are zero. The value of 'd' is arranged so as to minimise the value of 'e'. The guaranteed ranges for 'd' and 'e' are 17 digits and three digits respectively: e3 and d18, d17 may always be zero. A single precision number has a maximum exponent of 53 and 9 digits of significand; a double precision number has a maximum exponent of 340 and 17 digits of significance. The result when the packed values are &A through &F is undefined. Zero will always be stored as +zero, but either +0 or –0 may be loaded.

Floating point status register

There is a floating point status register (FPSR) which, like ARM's combined PC and PSR, has all the necessary status for the floating point system. The FPSR contains the IEEE flags but not the result flags – these are only available after floating point compare operations.

Each IEEE flag denotes a possible error condition. There is a corresponding 'trap' or interrupt enable flag for each one. If the trap is enabled, then the error condition will cause execution to stop with an error; otherwise a special result (eg not-a-number or infinity) is returned.

The flags contained in the status register are as follows:

IVO – invalid operation

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

- Any operation on something a NAN (not-a-number)
- Magnitude subtraction of infinities eg +infinity + –infinity
- Multiplication of 0 by an infinity
- Division of 0/0 or infinity/infinity
- x REM y where x is infinity or y is 0
- Square root of any number less than zero (but SQR(–0) is –0)
- Conversion to integer or decimal when overflow, infinity or operand not being a number make it impossible.
- Comparison with exceptions of unordered operands.
- ACS, ASN when argument's absolute value is > 1

- SIN, COS, TAN when argument is infinite
- LOG, LGN when argument <= 0

REM is the 'remainder after floating point division' operator.

DVZ – division by zero

If the divisor is zero and the dividend a finite, non-zero number then this exception occurs, or a correctly signed infinity is returned if the trap is disabled.

OFL – overflow

The OFL is set whenever the destination format's largest finite number is exceeded by the result after rounding has taken place. As overflow is detected after rounding a result, whether overflow occurs or not (after some operations) depends on rounding mode.

The untrapped result returned is the correctly signed infinity, independent of the rounding mode – overflow can be seen as a signal that an infinite result has been generated from an operation on finite values.

UFL – underflow

The UFL is set whenever a result is so tiny that it is rounded to zero, but has a non-zero value. As underflow is detected after rounding a result, whether underflow occurs or not after some operations depends on rounding mode.

The untrapped result returned is zero, with the sign set to that of the non-zero value.

INX – inexact

The INX is set if the rounded result of an operation is not exact (different from the value computable with infinite precision) or overflow has occurred while the OFL trap was disabled. If there is no trap the result will be used directly. OFL or UFL traps take precedence over INX. INX will also be set when computing SIN or COS or

TAN of values larger than 10^20 (ie values for which the multiple of PI ranging gives a useless answer).

For each flag, there are two bits of the instruction dedicated to it:

31......21	20	19	18	17	16	15......5	4	3	2	1	0
	INX	UFL	OFL	DVZ	IVO		INX	UFL	OFL	DVZ	IVO

Interrupt Masks Cumulative Flags

Whenever the appropriate condition arises, the cumulative flags in bits 0 to 4 are set. They can only become cleared by a WFS instruction. If the relevant interrupt mask is set, then the same condition that sets the cummulative flags also causes an exception to be delivered to the program. The floating point system provides the exception routine with a word indicating (in the same position as the cumulative flags) which floating point exception occured.

THE INSTRUCTION SET

Co-Processor data transfer

```
op<cond>prec  Fd, addr
```

`st fop`	is LDF for load, STF for store
`addr`	is [Rn]<, #offset> or [Rn ,#offset]<!>
`prec`	is the precision denoted by the letter S, D, E or P (see below)

The bit format of the instruction is

31...28	27...24	23	22	21	20	19...16	15...12	11...8	7......0
Cond	110P	U/D	Y	Wb	L/S	Rn	X Fd	0001	Offset

P	is pre- or post-indexed addressing
U/D	is positive/negative offset
YX	is the precision
Wb	is write-back (pre-indexed only)
L/S	is load or store
Rn	is the ARM base address register
Fd	is the FPU source/destination register
offset	is the scaled offset

Load (LDF) or store (STF) the high precision value into one of the four memory formats. On store, the value is rounded using the 'round to nearest' rounding method to the destination precision, or is precise if the destination has sufficient precision. Bits 22 and 15 are set from the precision letter, and determine the precision, as follows:

Precision	**Letter**	**Y**	**X**
Single	S	0	0
Double	D	0	1
Extended	E	1	0
Packed BCD	P	1	1

The offset is in words from the ARM base register, and is in the range –1020 to +1020. It is added to the base register in pre-indexed mode if write-back is specified, and always in post-indexed mode.

Co-Processor register transfer

```
FLT<cond>prec<round>        Fn,(Rd | #value)     0000
FIX<cond>prec<round>        Rd,Fn                0001
WFS<cond>                   Rd                   0010
RFS<cond>                   Rd                   0011
WFC<cond>                   Rd                   0100
RFC<cond>                   Rd                   0101
```

<round> is the optional rounding mode: P, M or Z; see below

FLT	Integer to Floating Point	Fn := Rd (or Fn := #value)	
FIX	Floating point to integer	Rd := Fm	
WFS	Write Floating Point Status	FPSR := Rd	
RFS	Read Floating Point Status	Rd := FPSR	
WFC	Write Floating Point Control	FPC := Rd	Supervisor Only
RFC	Read Floating Point Control	Rd := FPC	Supervisor Only

The binary format is:

31…28	27…24	23…21	20	19…16	15…12	11…8	7…4	3…0
Cond	1110	abc	L/S	e Fn	Rd	0001	fgh1	i Fm

abcL/S	are the operation code bits, as above (0110.. undefined)
ef	give the floating point precision, as above
gh	is the rounding mode, as below
Fm, Fn	are FPU register numbers
Rd	is an ARM register number
i	determined whether Fm is a register number of constant

The rounding modes are

Mode	Letter	e	f
Nearest		0	0
Plus infinity	P	0	1
Minus infinity	M	1	0
Zero	Z	1	1

The values allowed for immediate operands in FLT are:

Value	**Fm endcoding**
0.0	000
1.0	001
2.0	010
3.0	011
4.0	100
5.0	101
0.5	110
10.0	111

Constants cannot be specified in the Fn field for the FIX instruction since there is no point FIXing a known value into an ARM integer register. The MOV instruction should be used for this.

Co-Processor data operations

The formats of these instructions are:

```
binop<cond>prec<round>      Fd, Fn, (Fm|#value)
unyop<cond>prec<round>      Fd, (Fm|#value)
```

binop	is one of the binary operations listed below
unyop	is one of the unary operations.
Fd	is the FPU destination register
Fn	is the FPU source register (binops only)
Fm	is the FPU source register
#value	is the immediate operand, as an alternative to Fm

The binops are:

			opcode
ADF	Add:	Fd := Fn + Fm	00000
MUF	Multiply:	Fd := Fn * Fm	00010
SUF	Sub:	Fd := Fn – Fm	00100
RSF	Reverse Subtract:	Fd := Fm – Fn	00110
DVF	Divide:	Fd := Fm/Fn	01000
RDF	Reverse Divide:	Fd := Fm / Fn	01010
POW	Power:	Fd := Fn to the power of Fm	01100
RPW	Reverse Power:	Fd := Fm to the power of Fn	01110
RMF	Remainder	Fd := remainder of Fn / Fm	10000
FML	Fast Multiply:	Fd := Fn * Fm	10010
FDV	Fast Divide:	Fd := Fn / Fm	10100
FRD	Fast Reverse Divide:	Fd := Fm / Fn	10110
POL	Polar angle (ArcTan2):	Fd := polar angle of Fn, Fm	11000

The unops are:

MVF	Move:	Fd := Fm	00001
MNF	Move Negated:	Fd := – Fm	00011
ABS	Absolute value:	Fd := ABS (Fm)	00101
RND	Round to integral value:	Fd := integer value of Fm	00111
SQT	Square root:	Fd := square root of Fm	01001
LOG	Logarithm to base 10:	Fd := logten of Fm	01011
LGN	Logarithm to base e:	Fd := loge of Fm	01101
EXP	Exponent:	Fd := e to the power of Fm	01111
SIN	Sine:	Fd := sine of Fm	10001
COS	Cosine:	Fd := cosine of Fm	10011
TAN	Tangent:	Fd := tangent of Fm	10101
ASN	Arc Sine:	Fd := arcsine of Fm	10111
ACS	Arc Cosine:	Fd := arccosine of Fm	11001
ATN	Arc Tangent:	Fd := arctangent of Fm	11011

Note that wherever Fm is mentioned, a floating point constant could be used instead.

FML, FRD and FDV produce a result only accurate to single precision.

Final rounding is done only at the last stage of a SIN, COS etc – the calculations to compute the value are done with 'round to nearest' using the full working precision.

The binary format of the instruction is:

31...28	27...24	23...20	19...16	15...12	11...8	7...4	3...0
Cond	1110	abcd	e Fn	j Fd	0001	fgh0	i Fm

abcdj	is the opcode
ef	is the precision
gh	is the rounding mode
i	is 0 for Fm, 1 for immediate operand

Co-Processor Status Transfer

```
op<cond>prec<round>      Fm, Fn
```

op is one of the following:

CMF	Compare floating	compare Fn with Fm
CNF	Compare negated floating	compare Fn with –Fm
CMFE	Compare floating with exception	compare Fn with Fm
CNFE	Compare negated floating with exception	compare Fn with –Fm

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

The ARM flags N, Z, C, V refer to the following after compares:

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

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

ECONET, THE TRANSPORT LAYER

INTERFACES

There are only two interface styles to Econet, one is the simple case of *Help Station where the current station number and the network number (if known) are printed on the screen. The second style of interface is via a set of SWI calls. These SWI calls are listed below.

SWI Econet_CreateReceive

R0 => port
R1 => station
R2 => net
R3 => buffer address
R4 => size
R0 <= handle

SWI Econet_ExamineReceive

R0 => handle
R0 <= status

SWI Econet_ReadReceive

R0 => handle
R0 <= status
R1 <= control
R2 <= port
R3 <= station
R4 <= net
R5 <= buffer address
R6 <= size

SWI Econet_AbandonReceive

R0 => handle
R0 <= status

SWI Econet_WaitForReception

R0 => handle
R1 => delay
R2 => ESCapableFlag
R0 <= status
R1 <= control
R2 <= port
R3 <= station
R4 <= net
R5 <= buffer address
R6 <= size

Note that this interface enables interrupts and so cannot be called from within interrupt service code.

SWI Econet_EnumerateReceive

R0 => index
R0 <= handle

SWI Econet_StartTransmit

R0 => flag byte
R1 => port
R2 => station
R3 => net
R4 => buffer address
R5 => size
R6 => count
R7 => delay
R0 <= handle

SWI Econet_PollTransmit

R0 => handle
R0 <= status
R3 <= station
R4 <= net

SWI Econet_AbandonTransmit

R0 => handle
R0 <= status

SWI Econet_DoTransmit

R0 => flag byte
R1 => port
R2 => station
R3 => net
R4 => buffer address
R5 => buffer size
R6 => count
R7 => delay
R0 <= status
R2 <= buffer address
R3 <= station
R4 <= net

Note that this interface enables interrupts and so cannot be called from within interrupt service code.

SWI Econet_ReadLocalStationAndNet

R0 <= station
R1 <= net

Note that this interface enables interrupts and so cannot be called from within interrupt service code.

SWI Econet_ConvertStatusToString

R0 => status
R1 => buffer
R2 => size of the buffer
R3 => station number
R4 => network number
R0 <= buffer
R1 <= updated buffer
R2 <= updated size of the buffer

SWI Econet_ConvertStatusToError

R0 => status
R1 => pointer to error buffer
R2 => size of buffer
R3 => station number
R4 => network number
R0 <= standardErrorIndicator
Returns with V set

SWI Econet_ReadProtection

R0 <= mask word

SWI Econet_SetProtection

R0 => mask word

SWI Econet_ReadStationNumber

R1 => address of string to read
R1 <= address of terminating space or control character
R2 <= station number (–1 for not found)
R3 <= network number (–1 for not found)

SWI Econet_PrintBanner

SWI Econet_RegisterDomain

R0 <= domain number

SWI Econet_DeRegisterDomain

R0 => domain number

SWI Econet_AllocatePort

R0 <= port number

SWI Econet_DeAllocatePort

R0 => port number

SWI Econet_PreAllocatePort

R0 => port number

CONVENTIONS AND VALUES

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

Network numbers are normally in the range 0 to 254. The value zero in a CreateReceive is taken to indicate that reception may occur from any network. The network number 255 is used in StartTransmit and in DoTransmit to indicate that a broadcast is to take place; it is also used in CreateReceive to indicate that reception may occur from any station and is to be preferred over the value zero for this purpose.

Port numbers are normally in the range 1 to 254, although the values &90 through &9F, and &D0 through &D1 are reserved by Acorn for existing protocols. Port number zero is reserved in transmission to indicate that the transmission is an immediate operation. A port number of either zero or 255 in a reception indicates that the reception may occur regardless of the port number on the incoming packet.

Control values are in the range 128 (&80) to 255 (&FF).

Buffer addresses are any byte address and the size fields are always of the byte length.

All delays are in centi-seconds.

There are two ways of doing receptions, the first is for simple foreground processes:

```
SWI     XEconet_CreateReceive
LDRVC   r1, Delay
LDRVC   r2, EscFlag SWIVC
XEconet_WaitForReception
BVS     Error
TEQ     r0, Status_Received
BEQ     OK SWI
XEconet_ConvertStatusToError
BVS     Error ; Always taken
```

The second method involves the user, possibly even and interrupt or event processes, using the full four interfaces (Create, Examine, Read, and Abandon) in the usual way.

As with reception there are two ways for transmissions to take place, a simple way for use by foreground processes (DoTransmit), and a more complex method which is OK to use in the background. The background method involves starting the transmit process and then using the Poll interface to poll it to completion, then Abandoning it. Note that for both reception and transmission the call to Abandon is important in that it releases memory held for internal state.

All utilities and programs requiring the user to input a station number should use the SWI call to convert to numeric since this is then both efficient and consistant.

Outputting station numbers are handled by the MOS SWI calls XOS_ConvertFixedNetStation and XOS_ConvertNetStation.

MPLEMENTATION LIMITS

No immediate operations are possible from Arthur Econet.

Only the immediate operations 'Peek', 'Poke', and 'MachinePeek' are available to Arthur Econet.

None of the port allocation or domain registration SWIs are implemented.

INTERFACES

SWI Podule_ReadID

R3 => podule number
R0 <= podule ID byte

SWI Podule_ReadHeader

R2 => pointer to core, 8 or 16 bytes
R3 => podule number

SWI Podule_EnumerateChunks

R0 => chunk number (zero to start)
R3 => podule number
R0 <= next chunk number (zero for end)
R1 <= size in bytes
R2 <= type byte
R4 <= pointer to name if an RM else preserved

SWI Podule_ReadChunk

R0 => chunk number
R2 => pointer to core, assumed big enough
R3 => podule number

SWI Podule_ReadBytes

R0 => psuedo address
R1 => count in bytes
R2 => pointer to core
R3 => podule number

SWI Podule_WriteBytes

R0 => pseudo address
R1 => count in bytes
R2 => pointer to core
R3 => podule number

SWI Podule_CallLoader

R0 => user data
R1 => user data
R2 => user data
R3 => podule number
R0 <= user data
R1 <= user data
R2 <= user data

SWI Podule_RawRead

R0 => podule address (0..&3FFF)
R1 => count in bytes
R2 => pointer to core
R3 => podule number

SWI Podule_RawWrite

R0 => podule address (0..&3FFF)
R1 => count in bytes
R2 => pointer to core
R3 => podule number

APPENDIX A – ARM ASSEMBLER

INTRODUCTION

Assembly language is a programming language in which each statement translates directly into a single machine code instruction or piece of data. An assembler is a piece of software which converts these statements into their machine code counterparts.

Writing in assembly language has its disadvantages. The code is more verbose than the equivalent high-level language statements, more difficult to understand and therefore harder to debug. High-level languages were invented so that programs could be written to look more like English so we could talk to computers in our language rather than directly in its own.

Assembly language is used in preference to high-level languages. The first reason is that the machine code program produced by it executes more quickly than its high-level counterparts, particularly those in languages such as BASIC which are interpreted. The second reason is that assembly language offers greater flexibility. It allows certain operating system routines to be called or replaced by new pieces of code, and it allows greater access to the hardware devices and controllers.

USING THE BASIC ASSEMBLER

The assembler is part of the BBC BASIC language. Square brackets '[' and ']' are used to enclose all the assembly language instructions and directives and hence to inform BASIC that the enclosed instructions are intended for its assembler. However, there are several operations which must be performed from BASIC itself to ensure that a subsequent assembly language routine is assembled correctly.

Initialising external variables

The assembler allows the use of BASIC variables as addresses or data in instructions and assembler directives. For example variables can be set up in BASIC giving the numbers of any SWI routines which will be called. For example:

```
OS_WriteI = &100
..........
[
..........
SWI OS_WriteI+ASC">"
..........
```

Reserving memory space for the machine code

The machine code generated by the assembler is stored in memory. However, the assembler does not automatically set memory aside for this purpose. You must reserve sufficient memory by using the DIM statement. For example:

```
1000 DIM code% 100
```

The start address of the memory area reserved is assigned to the variable code%. The address of the last memory location is code%+100. Hence, it reserves a total of 101 bytes of memory.

Memory pointers

You need to tell the assembler the start address of the area of memory you have reserved. The simplest way to do this is to assign P% to point to the start of this area. For example:

```
DIM code% 100
..........
P% = code%
```

P% is then used as the program counter. The assembler places the first assembler instruction at the address P% and automatically increments the value of P% by four so that it points to the next free location. When the assembler has finished assembling the code, P% points to the byte following the final location used. Therefore, the number of bytes of machine code generated is given by:

```
P% - code%
```

This method assumes that you wish subsequently to execute the code at the same location.

The position in memory at which you load a machine code program may be significant. For example, it might refer directly to data embedded within itself, or expect to find routines at fixed addresses. Such a program only works if it is loaded in the correct place in memory. However, it is often inconvenient to assemble the program directly into the place where it will eventually be executed. This memory may well be used for something else whilst you are assembling the program. The solution to this problem is to use a technique called 'offset assembly' where code is assembled as if it is to run at a certain address but is actually placed at another.

To do this, set O% to point to the place where the first machine code instruction is to be placed and P% to point to the address where the code is to be run.

To notify the assembler that this method of generating code is to be used, the directive OPT, which is described in more detail below, must have bit 2 set.

Implementing passes

Normally, when the processor is executing a machine code program, it executes one instruction and then moves on automatically to the one following it in memory. You can, however, make the processor move to a different location and start processing from there instead by using one of the 'branch' instructions. For example:

```
.result_was_0
..............
BEQ result_was_0
```

The fullstop in front of the name result_was_0 identifies this string as the name of a 'label'. This is a directive to the assembler which tells it to assign the current value of the program counter (P%) to the variable whose name follows the fullstop.

BEQ means 'branch if the result of the last calculation was zero'. The location to be branched to is given by the value previously assigned to the label result_was_0.

The label can, however, occur after the branch instruction. This causes a slight problem for the assembler since when it reaches the branch instruction, it hasn't yet assigned a value to the variable, so it doesn't know which value to replace it with.

You can get around this problem by assembling the source code twice. This is known as two-pass assembly. During the first pass the assembler assigns values to all the label variables. In the second pass it is able to replace references to these variables by their values.

It is only when the text contains no forward references of labels that just a single pass is sufficient.

These two passes may be performed by a FOR...NEXT loop as follows:

```
DIM code% 400
FOR pass% = 0 TO 3 STEP 3
   P% = code%
   [
   OPT pass%
   ...
   ]
NEXT pass%
```

Note that the pointer(s), in this case just P%, must be set at the start of both passes.

The OPT is an assembler directive whose bits have the following meaning:

Bit	Meaning
0	Assembly listing enabled if set
1	Assembler errors enabled
2	Assembled code placed in memory at O% instead of P%

Bit 0 controls whether a listing is produced. It is up to you whether or not you wish to have one or not.

Bit 1 determines whether or not assembler errors are to be flagged or suppressed. For the first pass, bit 1 should be zero since otherwise any forward-referenced labels will cause the error `Unknown or missing variable` and hence stop the assembly. During the second pass, this bit should be set to one, since by this stage all the labels defined are known, so the only errors it catches are 'real ones'. For example, labels which have been used but not defined, or misspelt instructions, etc.

Bit 2 allows 'offset assembly', ie the program may be assembled into one area of memory, pointed to by O%, whilst being set up to run at the address pointed to by P%.

Executing a machine code program

Once an assembly language routine has been successfully assembled, the resulting machine code can be executed in a variety of ways:

```
CALL <address> or USR <address>
```

These may be used from inside BASIC to run the machine code at a given address. See the Archimedes **User Guide** for more details on these statements.

```
*<name> or *RUN <name> or */<name>
```

These will load and run the named file, using the locations defined when it was saved.

FORMAT OF ASSEMBLY LANGUAGE STATEMENTS

The assembly language statements and assembler directives should be between the square brackets.

There are very few rules about the format of assembly language statements, those which exist are given below:

- Each assembly language statement comprises an assembler mnemonic of one or more letters followed by a varying number of operands.
- Instructions should be separated from each other by colons or newlines.
- Any text following a full stop '.' is treated as a label name.
- Any text following a semicolon ';', or backslash '/', or 'REM' is treated as a comment and so ignored (until the next end of line or ':').
- Spaces between the mnemonic and the first operand, and between the operands themselves are ignored.

The BASIC assembler contains the following directives:

EQUB int	Define 1 byte of memory from LSB of int
EQUW int	Define 2 bytes of memory from int
EQUD int	Define 4 bytes of memory from int
EQUS str	Define 0 – 255 bytes as required by string expression str
ALIGN	Align P% (and O%) to the next word boundary
ADR reg,addr	Assemble instruction to load addr into reg

The first four operations initialise the reserved memory to the values specified by the operand. In the case of EQUS the operand field should be a string expression. In all other cases it may be a numeric expression. DCB, DCW, DCD and DCS are synonyms for these directives.

The ALIGN directive ensures that P% (and O% that is used) lies on a word boundary. It is used after, for example, an EQUS to ensure that the next instruction is word-aligned.

ADR assembles a single instruction, an ADD or SUB, with reg as the destination register. It obtains addr in that register in a PC-relative (ie position independent) manner.

REGISTERS

At any particular time there are sixteen 32-bit registers available for use, R0 to R15. However, R15 is special since it contains the program counter and the processor status register.

R15 is split up with 24 bits used as the program counter (PC) to hold the word address of the next instruction. 8 bits are used as the processor status register (PSR) to hold information about the current values of flags and the current mode/register bank. These bits are arranged as follows:

31..................................26 25...2 1 0

N	Z	C	V	I	F	PC (word aligned)	M

The top six bits hold the following information:

Bit	**Flag**	**Meaning**
31	N	Negative flag
30	Z	Zero flag
29	C	Carry flag
28	V	Overflow flag
27	I	Interrupt request disable
26	F	Fast interrupt request disable

The bottom two bits can hold one of four different values:

M	Meaning
0	User mode
1	Fast interrupt processing mode
2	Interrupt processing mode
3	Supervisor mode

User mode is the normal program execution state. Supervisor mode is a special mode which is entered when calls to the supervisor are made using software interrupts (SWIs) or when an exception occurs. From within supervisor mode certain operations can be performed which are not permitted in user mode, such as writing to hardware devices and peripherals. The supervisor has its own private registers R13 and R14. So after changing to supervisor mode, the registers R0 – R12 are the same, but new versions of R13 and R14 are available. The values, contained by these registers in user mode, are not overwritten or corrupted.

Similarly the interrupt and fast interrupt processing modes have their own private registers (R13 – R14 and R8 – R14 respectively).

Although only 16 registers are available at any one time, the processor actually contains a total of 27 registers.

CONDITION CODES

All the machine code instructions can be performed conditionally according to the status of one or more of the following flags: N, Z, C, V. The sixteen available condition codes are:

AL	Always	
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
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	Negative	N=1
NE	Not equal	Z=0
NV	Never	
PL	Positive	N=0
VC	Overflow clear	V=0
VS	Overflow set	V=1

Two of these may be given alternative names as follows:

LO	Lower unsigned is equivalent to CC
HS	Higher / same unsigned is equivalent to CS

THE INSTRUCTION SET

The available instructions are introduced below in categories indicating the type of action they perform and their syntax. The description of the syntax obeys the following standards:

< >	indicates that the contents of the brackets are optional
(x | y)	indicates the either x or y but not both may be given
#exp	indicates that an expression is to be used which evaluates to an immediate constant. An error is given if the value cannot be stored in the instruction.
Rn	indicates that an expression evaluating to a register number (in the range 0 – 15) should be used, or just a register name, eg R0. PC may be used for R15.

shift indicates that one of the following shift options should be used:

ASL	(Rn | #exp)	Arithmetic shift left by contents of Rn or expression
LSL	(Rn | #exp)	Logical shift left
ASR	(Rn | #exp)	Arithmetic shift right
LSR	(Rn | #exp)	Logical shift right
ROR	(Rn | #exp)	Rotate right
RRX		Rotate right one bit with extend

Arithmetic and logical instructions

Syntax: opcode<cond><S> Rd, <Rn>, (#exp | Rm <,shift>)

The instructions available are given below:

Instructions		**Result of opcode Rd, Rn, Rm**
ADC	Add with carry	Rd = Rn + Rm + C
ADD	Add without carry	Rd = Rn + Rm
SBC	Subtract with carry	Rd = Rn – Rm – (1 – C)
SUB	Subtract without carry	Rd = Rn – Rm
RSC	Reverse subtract with carry	Rd = Rm – Rn – (1 – C)
RSB	Reverse subtract without carry	Rd = Rm – Rn
AND	Bitwise AND	Rd = Rn AND Rm
BIC	Bitwise AND NOT	Rd = Rn AND NOT (Rm)
ORR	Bitwise OR	Rd = Rn OR Rm
EOR	Bitwise EOR	Rd = Rn EOR Rm
		Result of opcode Rd, Rm
MOV	Move	Rd = Rm
MVN	Move NOT	Rd = NOT Rm

Each of these instructions produces a result which it places in a destination register (Rd). The instructions do not affect bytes in memory directly.

As was seen above, all of these instructions can be performed conditionally. In addition, if the 'S' is present, they can cause the condition codes to be set or cleared. The condition codes N, Z, C and V are set by the arithmetic logic unit (ALU) in the arithmetic operations. The logical (bitwise) operations set N and Z from the ALU, C from the shifter and do not affect V.

Example:

```
ADDEQ R1, R1, #7         ; If the zero flag is set then add 7
                         ; to the contents of register R1.
```

Example:

```
SBCS R2, R3, R4          ; Subtract with carry the contents of
                         ; register R4 from the contents of
                         ; register R3 and place the result in
                         ; register R2. The flags will be
                         ; updated.
```

Example:

```
AND R3, R1, R2, LSR #2 ; Perform a logical AND on the contents
                       ; of register R1 and the contents of
                       ; register R2 * 4 and place the result
                       ; in register R3.
```

Special actions are taken if any of the source registers are R15; the action is as follows:

- If Rm=R15 all 32 bits of R15 are used in the operation ie the PC + PSR.
- If Rn=R15 only the 24 bits of the PC are used in the operation.

If the destination register is R15, then the action depends on whether the optional 'S' has been used:

- If S is not present only the 24 bits of the PC are set.
- If S is present the whole result is written to R15, the flags are updated from the result. (However the mode, I and F bits can only be changed when in non-user modes.)

Comparisons

Syntax: opcode<cond><P> Rn, (#exp | Rm <,shift>)

There are four comparison instructions:

Instruction		Calculation performed by opcode Rn, Rm
CMN	Compare	Rn + Rm
CMP	Compare	Rn – Rm
TEQ	Test equal	Rn EOR Rm
TST	Test	Rn AND Rm

These are similar to the arithmetic and logical instructions listed above except that they do not take a destination register since they do not return a result. Also, they automatically set the condition flags (since they would perform no useful purpose if they didn't). Hence, the 'S' of the arithmetic instructions is implied.

These routines have an additional function which is to set the whole of the PSR to a given value. This is done by using a 'P' after the opt code, for example CMNP.

Normally the flags are set depending on the value of the comparison. The I and F bits and the mode and register bits are unaltered. The 'P' option allows the corresponding eight bits of the result of the calculation performed by the comparison to overwrite those in the PSR (or just the flag bits in user mode). For example:

```
CMNP R0,#&FC000000  ; Set the PSR to the result of
                    ; R0 + &FC000000
```

In the above example, if R0 is previously set to zero then this instruction will perform the calculation 0 + &FC000000 = &FF000000. The top six bits of the result are, therefore, set and the bottom two bits are clear. Hence this will alter the PSR so that all the flags are set and user mode is selected. This example assumes that the instruction is used when it is privileged enough to change the mode bits, ie not in user mode.

Multiply instructions

Syntax: MUL<cond><S> Rd,Rm,Rs
MLA<cond><S> Rd,Rm,Rs,Rn

Instruction		Calculation performed
MUL	Multiply	Rd = Rm * Rs
MLA	Multiply-accumulate	Rd = Rm * Rs + Rn

The multiply instructions perform integer multiplication, giving the least significant 32 bits of the product of two 32-bit operands.

The destination register must not be R15 or the same as Rm. Any other register combinations can be used.

If the 'S' is given in the instruction, the N and Z flags are set on the result, the V flag is unaffected and the C flag is undefined.

Example: `MUL R1,R2,R3`

Example: `MLAEQS R1,R2,R3,R4`

Branching

Syntax: B(cond) expression
BL(cond) expression

There are essentially only two branch instructions but in each case the branch can take place as a result of any of the 16 condition codes:

B<cond>	Branch
BL<cond>	Branch and link

The branch instruction causes the execution of the code to jump to the instruction given at the address to be branched to. This address is held relative to the current location.

Example: `BEQ label1  ; branch if zero flag set`

Example: `BMI minus   ; branch if negative flag set`

The branch and link instruction performs the additional action of copying the address of the instruction following the branch, and the current flags, into register R14. R14 is known as the 'link register'. This means that the routine branched to can be returned from by transferring the contents of R14 into the program counter and can restore the flags from this register on return. Hence instead of being a simple branch the instruction acts like a subroutine call. For example:

```
BLEQ equal
          .........    ; address of this instruction
          .........    ; moved to R14 automatically

.equal    .........    ; start of subroutine
          .........

          MOV R15,R14  ; end of subroutine
```

Single register load/save instructions

Syntax: opcode<cond><B><T> Rd, address

The single register load/save instructions are as follows:

LDR	Load register
STR	Store register

These instructions allow a single register to load a value from memory or save a value to memory at a given address. Addresses are held in registers, whose names are enclosed in square brackets. The simplest form of address is a register number, in which case the contents of the register are used as the address to load from or save to.

Another option is to add the contents of another register, or an immediate value, to the contents of the first register. This sum is then used as the address. This is known as pre-indexed addressing since the address written to the register is calculated before the load/save takes place. The register can be optionally updated to contain the address which was actually used by adding a '!' after the closing square bracket.

Syntax	Address
[Rn]	Contents of Rn
[Rn,#m]<!>	Contents of Rn + m
[Rn,Rm]<!>	Contents of Rn + contents of Rm
[Rn,Rm,shift #s]<!>	Contents of Rn + (contents of Rm shifted by 's' places)

The alternative is post-indexed addressing. In this case the address being used is given solely by the contents of the register Rn. The rest of the instruction determines what value is written back into Rn. This write back is performed automatically; no '!' is needed. Post-indexing gets its name from the fact that the address written to the register is calculated after the load/save takes place.

Syntax	Value written back
[Rn],#m	Contents of Rn + m
[Rn],Rm	Contents of Rn + contents of Rm
[Rn],Rm,shift #s	Contents of Rn + (contents of Rm shifted by s places)

If the address is given as a simple expression, the assembler will generate a pre-indexed instruction using R15 (the PC) as the base register. If the address is out of the range of the instruction (+/– 4095 bytes), an error is given.

If the 'B' option is specified after the condition, only a single byte is transferred, instead of a whole word.

Multiple load/store instructions

Syntax: opcode<cond>(I | D)(A | B) Rn<!>, {Rlist}<^>

These instructions allow the loading or saving of several registers:

LDM	Load multiple registers
STM	Store multiple registers

The contents of register Rn give the base address from/to which the value(s) are loaded or saved. Rlist provides a list of registers which are to be loaded from or saved to. The order the registers are given, in the list, is irrelevant since the lowest numbered register will be loaded first and the highest one last. For example, a list comprising {R5,R3,R1,R8} will be loaded from/saved to in the order R1, R3, R5, R8, with R1 occupying the lowest address in memory.

The 'I' or 'D' indicates whether the addresses loaded from or saved to are to Increase or Decrease from the base address. Hence it is possible to load or save a series of registers from/to a base address and the locations above it or below it in memory.

The 'A' or 'B' indicates whether or not the addresses are to be altered After or Before each register is loaded or saved. If they are altered afterwards, then the first register is loaded from or saved to the base address, and the address will be either increased or decreased before the next load or save. If they are to be altered before, then the base address is either increased or decreased before the first load or save takes place, then again before the second, etc.

Example:

```
LDMIA R5, {R0,R1,R2}        ; where R1 contains the value &1484
                            ; This will load R0 from &1484
                            ;                R1 from &1488
                            ;                R2 from &148C
```

Example:

```
LDMDB R5, {R0,R1,R2}        ; where R1 contains the value &1484
                            ; This will load R0 from &1480
                            ;                R1 from &147C
                            ;                R2 from &1478
```

There are two further options. If a '!' is present after the register containing the base address then this register will be updated to contain the final address. In the two examples above this would leave R5 containing &1490 and &1478 respectively.

If a '^' is given at the end of the register list on a load, and R15 is contained in this list, then the whole 32 bits of R15 will be loaded, instead of just the program counter part. If R15 is not contained in the list then the user bank of registers is forced. On save, a '^' at the end of the list means force the use of the user bank.

The examples below show how a stack may be implemented. The stack is built downwards in memory and is a 'full' stack as opposed to an 'empty' one. In a full stack the stack pointer points to the address containing the last value added to the stack rather than to the address of the first available empty slot.

Example:
```
STMDB Stackpointer!, {R0,R1,R2,R3} ; push onto stack
```

Example:
```
LDMIA Stackpointer!, {R0,R1,R2,R3} ; pull from stack
```

To make it easier to use these instructions with stacks, alternative set of letters may be used after the instruction. These are 'F' for Full stack, ie one where the stack pointer holds the address of the last item to be pushed, or 'E' for Empty, where the SP holds the address of the next item to be pushed. The second letter may be 'D' for a descending stack, where the SP is decremented for a push and incremented for a pull, and 'A' for an ascending stack, where the opposite is true.

All Acorn software uses an FD (full, descending) stack, and you should too. The two examples above could be rewritten:

Example:
```
STMFD Stackpointer!, {R0,R1,R2,R3} ; push onto stack
```

Example:
```
LDMFD Stackpointer!, {R0,R1,R2,R3} ; pull from stack
```

SWI

Syntax: SWI <expression>

The SWI mnenomic stands for SoftWare Interrupt. On encountering a SWI, the CPU changes into Supervisor mode and stores the address of the next location in R14. Since this is written into the Supervisor's own copy of R14, the User value is not corrupted. The CPU then goes to the SWI routine handler via the hardware SWI vector containing its address.

The first thing that this routine does is to discover which SWI was requested. It finds this out by using the location addressed in R14 to read the current SWI instruction. The opcode for a SWI is 32 bits long; 4 bits identify the opcode as being for a SWI, 4 bits hold all the condition codes and the bottom 24 bits identify which SWI it is.

Hence 2^24 different SWI routines can be distinguished. When it has found which particular SWI it is, the routine executes the appropriate code to deal with it and then returns by placing the contents of R14 back into the PC and changing the caller's mode status.

See the chapter **FUNDAMENTAL OPERATING SYSTEM CONCEPTS** for a description of how the Arthur operating system handles SWIs, and the appendix **SUMMARIES OF OPERATING SYSTEM CALLS** for a list of the operating system SWIs.

APPENDIX B – THE LINKER

This appendix describes the operation of the linker.

USING THE LINKER

The Linker accepts as input one or more files written in the ARM Object Format; (AOF) resolves references between them, and produces an executable image. Some of the inputs accepted by link may be libraries of AOF files which are searched to resolve unresolved external references.

Link has two input requirements. These are:

- the name of the image to be produced
- the names of the files to be linked.

The image file is specified using the –image keyword. The files to be linked can be specified either as a comma-separated list using the –files keyword, or they may be contained in a file specified by the –via keyword. For example:

```
link -image test -files a,b,c
```

means take a, b and c as input, and produce an image file called test. (Note that there must be no spaces in a comma-separated list because space is a more powerful separator than comma.)

Alternatively, you can use:

```
link -image test -via myfiles
```

'myfiles' should contain a list of input files, one on each line. In this case, the explicit object file list may be omitted. For example, mylist might contain:

```
o.test
o.mylibrary/l
```

If both an object-file list and a –via file are specified, then the files listed in the argument to the –via keyword are simply appended to the object-file list. For example:

```
link -image test o.test1,o.test2 -via stdstuff
```

where stdstuff contains:

```
test3
testlib/l
$.arm.clib.o.ansilib/l
```

and has the same effect as:

```
link -image test o.test1,o.test2,test3 -lib testlib,$.arm.clib.o.ansilib
```

Any file in a list of files to be linked may be decorated with a '/l' denoting that it is a library and the –FIles keyword may be omitted. For example:

```
link -image test o.test,o.mylibrary/l
link -files o.test,o.mylibrary/l -image test
```

The order of arguments is not significant and, in general, neither is the order of files within a list of files, except that libraries are searched in the order they are listed. Keywords may be typed in upper or lower case and can be given in full or abbreviated. For example, you can use either of the following:

```
link -FI o.test,o.mylibrary/l -im test
link -fi o.test,o.mylibrary/l -image test
```

Optionally, a list of libraries may be given using the –LIBrary keyword. For example:

```
link o.hello -image p.hello -library $.arm.clib.o.ansilib,o.Arthurlib
```

When libraries are included in the list of object files, only the required parts of them will be included in the final image.

By default, link generates an 'ADFS' image, suitable for direct execution by the Command Line Interpreter. However, this is not suitable for the low-level debugger Dbug as it lacks the necessary symbolic data. To generate an image suitable for use with the Dbug tool, one of the –DBug or –AOF keywords must be specified. For example:

```
link -dbug -image test o.test1,o.test2 -via stdstuff
```

LINKER PRE-DEFINED SYMBOLS

The linker defines several useful symbolic values to which the assembly code programmer may refer. These may not be defined or redefined by the programmer. All these names begin with 'Image$$' and, indeed, use of all external symbol names beginning 'Image$$' is reserved to Acorn.

The names which may be relied on are as follows:

Image$$RO$$Base

Address of the first byte of the (notionally) read-only portion of the image.

Image$$RO$$Limit

Address of the first byte beyond the (notionally) read-only portion of the image.

Image$$RW$$Base

Address of the first byte of the read-write portion of the image.

Image$$RW$$Limit

Address of the first byte beyond the read-write portion of the image.

ADFS and AOF images are split into two areas: a notionally read-only area (which is write protected on systems with hardware which supports this) and a read-write area. Usually, the read-only area contains code and literal data and precedes the read-write area in the image so the whole image is bounded by Image$$RO$$Base and

Image$$RW$$Limit. It is not guaranteed that Image$$RO$$Limit = Image$$RW$$Base.

LINKER KEYWORDS

–image

The argument to this keyword is the filename of the resulting object file.

–files

The argument to this keyword is a comma-separated list of object files. The –files keyword may be omitted in which case the first and only (non keyword) positional argument is interpreted as the list of files.

–via

The argument to this keyword specifies a files from which a list of object file to link should be acquired. There should be one name per line in the file. This list is additional to and appended to that provided by the argument to the –files keyword.

–library

The argument to this keyword specifies a list of object file libraries in which to search for unresolved external symbols. Libraries are searched as many times as necessary to resolve external symbols.

–keep

This option is provided for backwards compatibility.

–adfs

This option specifies that the resulting file be suitable for direct execution by the Command Line Interpreter. In this case, the default base address is 32K. Under the Arthur operating system, –adfs is assumed by default.

Link also has numerous additional options reserved to Acorn. Some of these will be revealed by the command:

```
link -help
```

Most of these options are not relevant to the Archimedes system and can be ignored.

THE DEBUGGER

The debugger is for debugging AOF images not containing unresolved references within the execution path. It works by loading itself where the program would normally load, and loading the program further up in store. The program can be run at this higher address without the control of the debugger (in order to achieve the same placement if it should be important) using the utility m2run. The debugger takes control from the program by means of breakpoints, which cause a trap into the debugger. Thus you cannot use the debugger to stop a program which has gone wildly wrong and find out where it has gone wrong.

The debugger is only for use in user mode, as its entry method following a breakpoint causes SVC mode to be entered temporarily, thus corrupting SVC register 14.

The debugger is started by typing:

```
dbug   programname
```

where programname is a linked AOFimage. The debugger will respond with the prompt:

```
dbug:
```

whenever it is ready for input.

The debugger deals with input of commands, numbers, register names and other miscellaneous items. The following terms will be used later in this guide, and so are defined now.

- A name is a sequence of letters, digits, dot and dollar beginning with a letter.
- A decimal number is a non-empty sequence of decimal digits.
- A hexadecimal number is the '&' character followed by a possibly empty sequence of hexadecimal digits.
- A register name is one of: R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, PC (=R15), SL (=R13), SP (=R12), FP (=R11) and IP (=R10), LR (=R14).
- All register names refer to the user bank of registers.
- Expressions are composed of numbers, names, register names, brackets and +, –, /, *, ^ and '.'.

  The symbols +, –, / and * have their usual meanings, while:

  '.' refers to the last location examined or deposited into

  brackets may be used for clarity and forcing evaluation order

  ^ is a postfixed unary operator meaning 'the contents of' and delivers a word from a word-aligned address. In certain circumstances, a register name may be an expression on its own, such as when examining a register range. Otherwise, any register name used in an expression must be followed by ^ in order that the register contents be used. There is no meaning if the ^ is left out.

The debugger takes commands of the following forms:

Name arguments

For commands which are names, only the first character is significant and the case is ignored. Thus RUN, run and R are all equivalent.

Run

Syntax: R rest-of-line

The program is entered with rest-of-line as arguments which will be available to the argument decoder as if the program had been run from the supervisor command prompt. The debugger will not be entered again unless either a breakpoint is reached or a machine level trap of some sort is taken, such as an address exception.

Single step

Syntax: S

Executes the current instruction and then re-enters the debugger.

Continue

Syntax: C

Continues execution

The debugger re-enters the program. As with Run, the debugger can only regain control by means of a breakpoint or a machine level trap.

Quit

Syntax: Q

Leaves the debugger and returns to the supervisor.

Breakpoint

Syntax: B S location

B D location

B D

B L

Sets or deletes a breakpoint at the given location. The location may be an expression evaluating to a store address. A maximum of twenty breakpoints is permitted. Breakpoints work by replacing the existing instruction by an instruction which will cause a trap and enter the debugger. Breakpoints cannot be placed in ROM code. Deleting a breakpoint simply replaces the original instruction over the trap, and removes the breakpoint from the breakpoint table.

B D with no location specified will interactively delete all breakpoints.

B L will list the addresses of all breakpoints.

Unwind

Syntax: U

Unwinds the procedure call stack. This is only meaningful if the Acorn procedure call standard has been used for procedure calling within the program you are debugging. In this case it will give a list of procedure calls made starting with the most recent and ending at the mainline code. (See the appendix **ACORN PROCEDURE CALL STANDARD.**)

List

Syntax: L name

Lists all symbols in the symbol table from the AOF which start with the given name. The output is not sorted and is, therefore, in the order in which the symbols were encountered in the AOF file.

Examine

Syntax: E Format

E a1 Format

E a1:a2 Format

E a1,a3 Format

Examines locations and produces output in format or in the default format if it is omitted. a1 is the first location examined; if omitted then the location after the last location examined/deposited into will be examined. Examination will continue until location a2 or for a3 locations, or if these are omitted only one location will be examined. The output format is specified below.

	Address	**Value**
Base	Current base	As specified
Style	Symbolic	As specified
Length	4	As specified

Deposit

Syntax: D a1 value

Deposits value in location a1 or if a1 is omitted then in the next location, that is the one after the last to be examined/deposited into.

Specifies the amount of store to be updated (byte, half-word or word).

Convert

Syntax = a1

Converts (display) a1 in given format, or in the default format if none has been given.

Format

Sets default format for output. The formats are:

Sizes

$W = word
$H = halfword
$B = byte

Bases

$D = decimal
$X = hexadecimal
$R = number in base 'r'

Styles

$C = character
$I = instruction
$S = string
$Y = symbolic name
$N = numeric,
$Z = signed numeric

The default format is $W $X $I.

APPENDIX C – ARM PROCEDURE CALL STANDARD

INTRODUCTION

This document relates to compiler implementation on the ARM. The reader should be familiar with the ARM's instruction set [ARM], floating point instruction set [AFP] and assembler syntax [AASM] before attempting to use this information to implement code generators for the ARM. In order to write a run-time system for a language implementation, additional information specific to the operating system will be necessary.

The main topics described herein are the procedure call and stack disciplines. These methodologies are followed in all Acorn language implementations for the ARM. The usefulness of any new language implementation will be directly related to the degree of compatibility between that language and those provided by Acorn.

At the end of this document are several examples of the usage of the standard with suggestions for generating effective code for the ARM.

Goals

The reduced instruction set of the ARM does not include a procedure call instruction, but a set of rules has been devised to facilitate calls between languages on the ARM, and the porting of language implementations between operating systems. These rules are used by the C and Modula–2 Plus compilers developed for use on the ARM, and other language implementors are strongly encouraged to use them, too.

The standard defines the use of registers and the passing of arguments at an external procedure call, and the format of a data structure that can be used by stack backtrace programs in reconstructing a sequence of outstanding calls.

The standard only defines what happens when an 'm' external procedure call occurs. Languages may choose to use other mechanisms for internal calls, and are not required to follow the register conventions described in this document except at the instant of an external call or return.

Design criteria

The procedure call standard was produced after a great deal of experimentation and study of other architectures and is believed to be the best possible compromise between various requirements. The following important factors influenced this design.

- The procedure call must be extremely fast.
- The call sequence must be as compact as possible. (Code density on RISC machines is a well-known problem. In typical compiled code, calls are believed to outnumber entries by about 5 to 1. The ARM instruction set is considerably more efficient when working out of registers than out of memory, so as many operations as possible must take place in registers. This is true to a certain extent on most machines, but substantially more so on the ARM than, say, on a VAX.)
- When multiple threads of control are being used within one address space, a separate stack is needed for each thread of control. Rather than having to specify the stack size of each thread at creation time, the standard is devised so that the stack can be extended in a non-contiguous manner, in 'm' stack chunks.
- The standard should encourage the production of re-entrant programs, with writable data separated from code.
- To accomodate analysis or change of procedure calls, other than the conventional return of outstanding called procedures, the design requires support of a 'stack backtracking' technique. Examples include debuggers providing information about local variables, C LongJmp and Modula–2 Plus exceptions. The procedure call standard defines enough about stack structure to ensure that these are possible, within certain stated limits.

The following names are used when referring to ARM registers:

ARM Register Names

```
a1      RN      0       ; argument 1/integer result
a2      RN      1       ; argument 2
a3      RN      2       ; argument 3
a4      RN      3       ; argument4
v1      RN      4       ; register variable
v2      RN      5       ; register variable
v3      RN      6       ; register variable
v4      RN      7       ; register variable
v5      RN      8       ; register variable
v6      RN      9       ; register variable
fp      RN      10      ; frame pointer
ip      RN      11      ; used as temp workspace
sp      RN      12      ; lower end of current stack frame
sl      RN      13      ; stack limit
lr      RN      14      ; link address on calls/workspace
pc      RN      15      ; program counter and processor status

f0      FN      0       ; floating point result
f1      FN      1       ; floating point scratch register
f2      FN      2       ; floating point scratch register
f3      FN      3       ; floating point scratch register
f4      FN      4       ; floating point preserved register
f5      FN      5       ; floating point preserved register
f6      FN      6       ; floating point preserved register
f7      FN      7       ; floating point preserved register
```

Please note that references to 'the stack' denoted by sp assume a stack that grows from high memory to low memory, with sp pointing at the top (i.e., lowest addressed) word in the stack.

For any register 'r', the phrase 'in r' in the following text refers to the contents of 'r'. The phrase 'at [r]' or 'at [r, #n]' refers to the word pointed at by 'r' or r+n, in line with the corresponding assembler syntax.

Data Representation and Argument Passing

This standard does not describe the layout in store of records, arrays and so forth, used by C and Modula–2 Plus on the ARM. For this information, consult the Acorn documentation of each language. The procedure call standard is defined in terms of m n word-sized arguments being passed from the caller to the callee, and a single word or floating point result that is passed back by the callee. For a detailed description of how these facilities are used to implement open array arguments, structure arguments, structure results, etc., also consult the Acorn documentation for each language.

Using Registers and Argument Passing in External Procedures

m Control Arrival

At the instant when control arrives at the target procedure, the following statements should be true. [For any m m, if a statement is made about argm and m n<m, then the statement can be ignored.]

- arg1 is in a1.
- arg2 is in a2.
- arg3 is in a3.
- arg4 is in a4.
- arg5 is at [sp].
- for all m > 5, argm should be at [sp, #4*(m–5)].
- fp contains 0 or points to a backtrace structure, as described in the next section.
- The values in sp, sl, fp are all multiples of 4.
- sp+256 >= sl.

- The values of sp and sl are such that all words below the word at [sp] and above or including the word at [sl, #–512] are readable, writable memory which can be used by the called procedure as temporary workspace and be overwritten with any values before the return of this procedure.

- lr contains the pc+psw value that should be restored into r15 on exit from the procedure. This is known as the 'm' return link value for this procedure call.

- pc contains the entry address of the target procedure.

- The value in sl is a 'm' stack chunk handle. This concept is only relevant if stack extension is being used, and its exact meaning depends on the operating system or language run-time system in use. It provides enough information to ensure that the stack can be extended with an extra chunk if necessary.

m Return Link

At the instant when the return link value for a procedure call is placed in the pc+psw, the following statements should be true.

- fp, sp, sl, v1, v2, v3, v4, v5, v6, f4, f5, f6 and f7 should contain the same values as they did at the instant of the call.

- If the procedure returns a word-sized result, R, which is not a single-precision floating point value, then 'r' should be in a1.

- If the procedure returns a single or double precision floating point result, fpr, then fpr should be in f0.

Notes

- The requirements of C preclude the passing of floating point arguments in floating point registers.

- The values of a2, a3, a4, ip and lr are not defined at the instant of return. The maintenance of the registers v1 to v6, however, suggests that this should be thought of as a 'callee-saving' standard.

- The values of the Z, N, C and V flags are loaded from the corresponding bits in the return link value on procedure return. This means, in the case where a procedure is called using a BL instruction, that these flag values will be preserved across the call.

- The values of fp and sp are not defined at arbitrary execution moments during the evaluation of a procedure, only at the instants of call and return.

- The minimum amount of stack defined to be available is not particularly great, and as a general rule a language implementation should not expect much more than this. Code generated by the C and Modula–2 Plus compilers, if there is inadequate local workspace, is able to allocate more stack space from the storage allocator and continue operation. Any language unable to do this may have its interaction with C and Modula2 impaired. The fact that sl contains a stack chunk handle may be important in achieving this.

- The statements about sp and sl are designed to optimise the testing of the one against the other. It is anticipated that a procedure's entry sequence might include something like:

```
CMP     sp, sl
BLLO    AllocateNewStackChunk
```

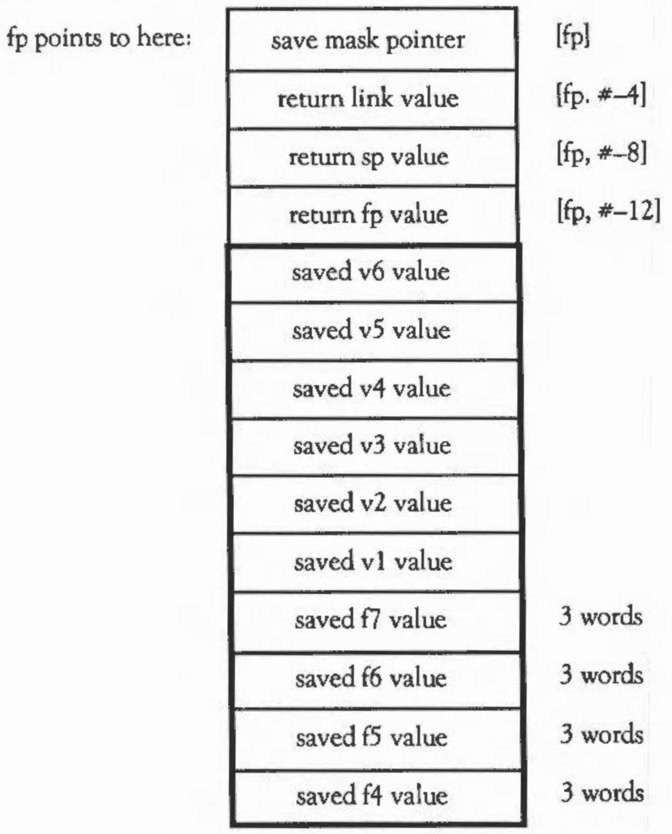
where AllocateNewStackChunk is a part of the run-time system for that language. If this test fails, and AllocateNewStackChunk is not called, then:

There are at least 512 bytes free on the stack.

This procedure should only call other procedures when sp has been dropped by 256 bytes or less. This will guarantee that there is enough space for the called procedure's entry sequence to work in.

If these limits are not enough, then the entry sequence may have to drop sp before performing the test.

At the instant of an external procedure call, the value in fp is zero or it points to a data structure that gives information about the sequence of outstanding procedure calls. This structure is in the following format:



The diagram shows between four and ten word-sized values, with those higher on the page being at the higher address in memory. The lowest ten values are entirely optional, and the presence of any does not imply the presence of any other. The floating point values are in extended format and occupy three words each. At the instant of procedure call, all of the following statements about this structure must be true:

The 'm' return fp value in the diagram is either 0 or contains a pointer to another stack backtrace data structure of the same form. Each of these corresponds to an active, outstanding procedure invocation. The same statements listed here are just as true about this next stack backtrace data structure as they are for the current one. Thus, the statements hold true for each structure in the chain.

The 'm' save mask pointer value, when bits 0, 1, 26, 27, 28, 29, 30, 31 have been cleared, points twelve bytes beyond a word known as the 'm' return data save instruction.

The return data save instruction is a word that corresponds to an ARM instruction of the following form:

```
STMDB   sp!,{[a1],[a2],[a3],[a4],[v1],[v2],[v3],[v4],[v5],[v6],fp,ip,lr,pc}
```

Note the square brackets in the above form denote optional parts: thus, there are 1024 possible allowable values for the return data save instruction, corresponding to the following bit patterns:

```
1110 1001 0010 1100 1100 11xx xxxx xxxx
```

The lowest 10 bits represent the registers, i.e. if bit 'n' is set, then register m n will be transferred.

The optional parts [v1], [v2], [v3], [v4], [v5] and [v6] in this instruction correspond to those optional parts of the stack backtrace data structure that are present such that: for all m m, if [vm] is present then so is [| saved vm value |], and if [vm] is absent then so is [| saved vm value |]. This is just as though the stack backtrace data structure was formed by the execution of this instruction, following the loading of ip from sp – as is very probably the case. Nothing should be deduced from the presence or absence of the optional parts [a1], [a2], [a3], [a4].

The sequence of up to four instructions following the return data save instruction decides if the saved floating point registers are present. The four instructions that are allowed in this sequence are:

```
STFE     f7, [sp, #-12] ; [f7], xxxxxxxx xxxxxxxx xxxxxxx xxxxxxxx
STFE     f6, [sp, #-12] ; [f6]
STFE     f5, [sp, #-12] ; [f5]
STFE     f4, [sp, #-12] ; [f4]
```

Any or all of these instructions may be missing, and any deviation from this order or any other instruction terminates the sequence.

The optional instructions [f4], [f5], [f6] and [f7] in this sequence correspond to those optional parts of the stack backtrace data structure that are present such that: for all m m, if [fm] is present then so is [| saved fm value |], and if [fm] is absent then so is [| saved fm value |]. This is just as though the stack backtrace data structure was formed by the execution of this sequence, as is probably the case.

At the instant when procedure a calls procedure b, the stack backtrace data structure pointed at by fp contains exactly those elements [v1], [v2], [v3], [v4], [v5], [v6], [f4], [f5], [f6], [f7], fp, sp and pc which must be restored into the corresponding ARM registers in order to cause a correct exit from procedure a, albeit with a junk result.

The following example suggests what the entry and exit sequences for a procedure are likely to be. Though not defined in terms of the following sequences because that would be unnecessarily restrictive, the entry sequence to a typical procedure might be expected to look something like:

```
MOV      ip, sp
STMDB    sp!, {args, workspace, fp, ip, lr, pc}^
SUB      fp, ip, #4
```

The corresponding exit sequence would be:

```
LDMDB    fp, workspace, fp, sp, pc^
```

Many apparent idiosyncrasies in the standard may be explained by efforts to make the entry sequence work smoothly. The example above is neither complete (no stack limit checking) nor mandatory (making arguments contiguous for C, for instance, requires a slightly different entry sequence).

The 'workspace' registers mentioned above correspond to as many of v1 to v6 that this procedure needs in order to work smoothly. At the instant when procedure a calls any other, those registers not mentioned in a's return data save instruction will contain the values that they contained at the instant that a was entered. Here is some sample assembly code as it might be produced by the C compiler:

```
; gggg is a function of 2 args that needs one register variable (v1)

gggg   MOV     ip, sp
       STMDB   sp!, {a1, a2, v1, fp, ip, lr, pc}
       SUB     fp, ip, #4          ; points at saved PC
       CMPS    sp, sl
       BLLO    stack_overflow|     ; handler procedure
       ....
       MOV     v1, ...             ; use a register variable
       BL      ffff|
       ...     v1 ...              ; rely on its value after ffff()
       ...
```

Within the body of the procedure, arguments are used from registers, if possible; otherwise they must be addressed relative to fp. In the two-argument case shown above, arg1 is at [fp,##–24] and arg2 is at [fp,##–20], but as discussed below, args will sometimes be stacked with positive offsets relative to fp. Local variables are never addressed offset from fp, they always have positive offsets relative to sp. In code that changes sp, this means that the offsets used may vary from place to place in the code. The reason for this is that it permits the procedure at stack_overflow to recover by setting sp (and sl) to some new stack segment as necessary. As part of this mechanism, stack_overflow may alter memory offset from fp by negative amounts, e.g. [fp, #–64] and downwards, provided that it adjusts sp to provide workspace for the called routine. If the function is going to use more than 256 bytes of stack it must go:

```
    SUB     ip, sp, #<my stack size>
    CMPS    ip, sl
    BLLO    stack_overflow_1|
```

instead of the two-instruction test shown above.

If a function expects no more than 4 arguments it can push all args onto the stack at the same time as saving its old fp and its return address (see the example above), and arguments are then saved contiguously in memory with arg1 having the lowest address. A function that expects more than 4 arguments has code at its head:

```
MOV      ip, sp
STMFD    sp!, {a1, a2, a3, a4}
STMFD    sp!, {v1, v2, fp, ip, lr, pc}    ; v1-v6 saved as necessary
SUB      fp, ip, #20                      ; point at saved PC
CMPS     sp, sl
BLLO     stack_overflow|
...
...
LDMDB    fp, {v1, v2, fp, sp, pc}^        ; restore register vars & return
```

where the header arranges that arguments (however many there are) lie in consecutive words of memory, and the return sequence that sp is always the lowest address on the stack that still contains useful data.

The time taken for a call, enter and return, with no arguments and no registers saved, is about 22 S-cycles, less than 3 microseconds on an uninterrupted ARM2.

ARM2 is Acorn's second-generation ARM processor.

Modula–2 Plus has slightly different requirements from C. For instance, there is no requirement for arguments to be contiguous, but on the other hand (due to open array arguments) the size of stack frames is not always computable at compile time. The following entry sequence is used in the presence of 4 or less arguments:

```
MOV      ip, sp
STMDB    sp!, {args, workregs, fp, ip, lr, pc}
SUB      fp, ip, #4
SUB      sp, sp, #workspace
CMP      sp, sl
BLLO     SYSTEM.StackOverflow
```

Note that the arguments are not necessarily stored in the stack frame: if they are frequently referenced in the procedure (and not used as VAR arguments to further calls) it may be more efficient to transfer them into preserved work registers soon after the entry sequence.

The actions performed at SYSTEM.StackOverflow are as follows:

– If there isn't another stack chunk forward chained onto this one, then allocate a new legal stack chunk.

– Change the values in fp and sp so that they are the same distance apart, pointing near the high address end of the new stack chunk. Copy 14 words from the place fp used to point, to where it now does. This represents the maximum number of registers that could have been saved, exclusive of sp and sl). The copy will be used as the stack backtrace data structure.

It might seem that this copy is wasteful, but the alternative (in the presence of variable sized stack frames) is to use an extra register (in addition to fp and sp) to denote the stack frame. The copy is actually quite quick, since it is of a known, fixed size. Note that the standard allows C and Modula2 the freedom to use slightly different stack extension routines, in order to optimise the most likely path in each case.

In the presence of 5 or more arguments, the arguments are copied if a stack extension occurs. This prevents having to allocate a whole register to this task.

Example:

```
ap  RN      v6

    MOV     ip, sp
    STMDB   sp!, {args, workregs, ap, fp, ip, lr, pc}
    SUB     fp, ip, #4
    SUB     sp, sp, #workspace
    CMP     sp, sl
    MOVLO   ip, #nargs
    BLLO    SYSTEM.StackOverflowN|
```

The exit sequence is the same as for C.

Although not required by the standard, the values in fp and sp are maintained while executing code produced by both the Modula–2 Plus and the C compilers. This makes it much easier to debug compiled code.

The following Modula–2 Plus types are implemented as words:

BITSET
BOOLEAN
CARDINAL
INTEGER
REAL POINTER TO (anything)
REF (anything)
REFANY
SET OF (anything)
SYSTEM.ADDRESS
SYSTEM.WORD

Variables and record fields of type CHAR are implemented using words, but an ARRAY OF CHAR is implemented using bytes. The values used for BOOLEAN values are: 0=FALSE and 1=TRUE.

The Modula–2 Plus type SystemTypes.String is implemented as a C-style string,

in order to aid interaction between the languages. That is, it is a pointer to a zero-terminated sequence of characters. There is no alignment or padding requirement for either the beginning or the end of the string. Most string manipulation in Modula–2 Plus programs is done in terms of the type Strings.String, rather than ARRAY OF CHAR.

The C and Modula–2 Plus concepts of pointer, record and array are broadly similar, and so most structures can be shared successfully between the two languages. The same is not quite true, however, of multi-word arguments that are passed by value.

The Modula–2 Plus compiler goes to considerable trouble to optimise cases whereby a multi-word value such as a structure or array is passed by value (in theory), but where passing by reference is in fact adequate. Examples are where an object passed by value is never written to, or is only used to pass on to other procedures. The frequent use of ARRAY OF CHAR parameters, for instance, makes this an extremely worthwhile optimisation. The method that Modula–2 Plus uses is that any multi-word object passed to a Modula–2 Plus procedure (except double precision real, see below) is in fact passed by reference. If the called procedure wishes to update the object then it copies it into its own stack frame as part of the entry sequence to the procedure. The C language does not do this – all arguments in C are passed by value and all types, including structures and arrays, are copied into the argument list.

A Modula–2 Plus open array argument is represented as two word-sized arguments, a pointer to the first element of the array and the number of elements in the array.

Cases where this impedes the passing of objects between C and Modula-2 Plus are very rare and can always be circumvented by the construction of appropriate type definitions in the two languages for a procedure.

Double precision reals are an exception to this rule, and are passed by value in both languages.

Multi-word results other than double precision reals in C and Modula–2 Plus programs are represented as an implicit first argument to the call, which points to where the caller would like the result placed. It is the first, rather than the last, so that it works with a C program that is not given enough arguments.

The procedure call standard is reasonably easy and natural for assembler programmers to use. It is encapsulated in a collection of macros that help assembler programmers to conform to the standard and construct the extra data structures needed for pseudo-Modula–2 Plus modules.

The Acorn Fortran–77 compiler uses calls that conform to this standard, except that, owing to the tendency for all Fortran–77 arguments to be pointers that can be computed statically, all calls are compiled as calls with a single argument, which points to a statically constructed argument pointer record. Thus, C and Modula–2 Plus can call Fortran routines, provided they make the relevant data structure definitions, but Fortran cannot call general C and Modula–2 Plus routines directly.

Since all Fortran I/O is done through built-in language features rather than through general procedure calls, this ability is worth sacrificing.

Note that there is no requirement specified by the standard concerning the production of re-entrant code, as this would place an intolerable strain on the conventional programming practices used in C and Fortran. The performance of a procedure in the face of multiple overlapping invocations is part of the specification of that procedure.

All of these languages have their own special requirements that make it inappropriate to use a procedure call of the form described here internally. All are capable of making external calls of the given form, through a small amount of assembler 'glue' code.

This document is not intended as a general guide to the writing of code generators, however it seems worthwhile to highlight various optimisations that appear particularly relevant to the ARM and to this standard.

The standard uses callee-saving rather than caller-saving because of the low cost of a LDM which loads many registers, and after statistical analysis of the code generated by both caller- and callee-saving code generators. The caller-saving code was found to be somewhat bulkier, with a higher proportion of LDM and STM instructions.

The preservation of condition codes over a procedure call is often useful because any short sequence of instructions (including calls) that forms the body of a short IF statement can be executed without a branch instruction.

For example:

```
    IF a < 0 THEN b := Foo(); END;
```

can compile into:

```
    CMP     a, #0
    BLLT    Foo
    MOVLT   b, a1
```

In the case of a 'leaf' or 'fast' procedure, i.e., one that calls no other procedures, much of the standard entry sequence can be omitted. In very small procedures, such as are frequently used in data abstraction modules, the cost of the procedure can be very small indeed. For instance, consider:

```
TYPE Foo = POINTER TO FooRecord;
   FooRecord = RECORD ...; bar : Bar; ... END;

PROCEDURE GetBar(foo : Foo): Bar; BEGIN RETURN foo^.bar; END GetBar;
```

The procedure GetBar can compile to just:

```
   LDR      a1, [a1, #barOffset]
   MOVS     pc, lr
```

This is also useful in procedures with a conditional as the top level statement, where one or other arm of the conditional is 'fast' – it calls no procedures. In this case there is no need to form a stack frame there.

For example, using this, the Modula–2 Plus program:

```
PROCEDURE Sum(i : INTEGER);
   BEGIN IF i <= 1 THEN RETURN i; ELSE RETURN i + Sum(i-1); END; END Sum;
```

could compile into:

```
   CMP      a1, #1    ; try fast case
   MOVSLE   pc, lr    ; and if appropriate, handle quickly!

   ; ELSE, form a stack frame and handle the rest as normal code.
   MOV      ip, sp
   STMDB    sp!, {v1, fp, ip, lr, pc}
   CMP      sp, sl
   BLLO     overflow

   MOV      v1, a1                    ; register to hold i
   SUB      a1, a1, #1                ; set up argument for call
```

```
BL      prod                    ; do the call
ADD     a1, a1, v1              ; perform the addition
LDMDB   fp, {v1, fp, sp, pc}^   ; and return
```

This is only worthwhile if the test can be compiled using only ip, and any spare of a1, a2, a3 and a4, as scratch registers. This technique could easily have a significant impact on certain speed-critical routines, such as read and write character.

For information on other publications, please contact Customer Services at the address given at the beginning of this manual.

APPENDIX D – OPERATING SYSTEM CALLS

INDEX OF SWI CALLS

SWI name	Number	Page
OS_WriteC	&00	51
OS_WriteS	&01	51
OS_Write0	&02	51
OS_NewLine	&03	52
OS_ReadC	&04	139
OS_CLI	&05	184
OS_Byte	&06	13
OS_Word	&07	13
OS_File	&08	233
OS_Args	&09	252
OS_BGet	&0A	251
OS_BPut	&0B	251
OS_GBPB	&0C	244
OS_Find	&0D	242
OS_ReadLine	&0E	141
OS_Control	&0F	336
OS_GetEnv	&10	338
OS_Exit	&11	335
OS_SetEnv	&12	339
OS_IntOn	&13	35
OS_IntOff	&14	35
OS_CallBack	&15	340
OS_EnterOS	&16	35
OS_BreakPt	&17	341
OS_BreakCtrl	&18	342
OS_UnusedSWI	&19	342
OS_UpdateMEMC	&1A	330
OS_SetCallBack	&1B	341
OS_Mouse	&1C	134
OS_Heap	&1D	326
OS_Module	&1E	358
OS_Claim	&1F	17
OS_Release	&20	18

SWI name	Number	Page
OS_ReadUnsigned	&21	403
OS_GenerateEvent	&22	37
OS_ReadVarVal	&23	346
OS_SetVarVal	&24	347
OS_GSInit	&25	411
OS_GSRead	&26	412
OS_GSTrans	&27	414
OS_BinaryToDecimal	&28	407
OS_FSControl	&29	294
OS_ChangeDynamicArea	&2A	331
OS_GenerateError	&2B	11
OS_ReadEscapeState	&2C	174
OS_EvaluateExpression	&2D	404
OS_SpriteOp	&2E	422
OS_ReadPalette	&2F	121
OS_ServiceCall	&30	364
OS_ReadVduVariables	&31	122
OS_ReadPoint	&32	124
OS_UpCall	&33	19
OS_CallAVector	&34	18
OS_ReadModeVariable	&35	124
OS_RemoveCursors	&36	126
OS_RestoreCursors	&37	127
OS_SWINumberToString	&38	410
OS_SWINumberFromString	&39	410
OS_ValidateAddress	&3A	332
OS_CallAfter	&3B	394
OS_CallEvery	&3C	395
OS_RemoveTickerEvent	&3D	395
OS_InstallKeyHandler	&3E	174
OS_CheckModeValid	&3F	127
OS_ChangeEnvironment	&40	343
OS_ClaimScreenMemory	&41	322
OS_ReadMonotonicTime	&42	394
OS_SubstituteArgs	&43	414

SWI name	Number	Page
OS_PrettyPrint	&44	52
OS_Plot	&45	53
OS_WriteN	&46	53
OS_ConvertStandardDateAndTime	&C0	400
OS_ConvertDateAndTime	&C1	401
OS_ConvertHex1/2/4/6/8	&D0 – &D4	408
OS_ConvertCardinal1/2/3/4	&D5 – &D8	408
OS_ConvertInteger1/2/3/4	&D9 – &DC	408
OS_ConvertBinary1/2/3/4	&DD – &E0	408
OS_ConvertSpacedCardinal1/2/3/4	&E1 – &E4	409
OS_ConvertSpacedInteger1/2/3/4	&E5 – &E8	409
OS_ConvertFixedNetStation	&E9	409
OS_ConvertNetStation	&EA	409
OS_WriteI	&100	53

NUMERIC INDEX OF OS_BYTE CALLS

OS_Byte	Description	Page
&CB (203)	Read/write RS423 input buffer minimum space	180
&CC (204)	Read/write RS423 ignore flag	180
&D3 (211)	Read/write bell channel	114
&D4 (212)	Read/write bell sound volume	114
&D5 (213)	Read/write bell frequency	114
&D6 (214)	Read/write bell duration	115
&D8 (216)	Read/write length of function key string	164
&D9 (217)	Read/write paged mode line count	115
&DA (218)	Read/write bytes in VDU queue	115
&DB (219)	Read/write Tab key code	164
&DC (220)	Read/write escape character	165
&DD (221)	Read/write interpretation of input values &C0 – &CF	165
&DE (222)	Read/write interpretation of input values &D0 – &DF	165
&DF (223)	Read/write interpretation of input values &E0 – &EF	165
&E0 (224)	Read/write interpretation of input values &F0 – &FF	165
&E1 (225)	Read/write function key interpretation	167
&E2 (226)	Read/write Shift function key interpretation	167
&E3 (227)	Read/write Ctrl function key interpretation	167
&E4 (228)	Read/write Ctrl Shift function key interpretation	167
&E5 (229)	Read/write Escape key status	169
&E6 (230)	Read/write escape effects	169
&EC (236)	Read/write character destination status	55
&ED (237)	Read/write cursor key status	149
&EE (238)	Read/write numeric keypad interpretation	170
&F0 (240)	Read country flag	49
&F1 (241)	Read/write user flag	48
&F2 (242)	Read RS423 baud rates	180
&F5 (245)	Read printer driver type	61
&F6 (246)	Read/write printer ignore character	62
&F7 (247)	Read/write Break key actions	172
&FA (250)	Read VDU driver screen bank number	107
&FB (251)	Read display screen bank number	108
&FD (253)	Read last break type	172
&FE (254)	Set effect of Shift Ctrl on numeric keypad	173
&FF (255)	Read/write boot option	232

NUMERIC INDEX OF OS_WORD CALLS

INDEX