Ursula
Kernel Functional Specification

EXPURGATED

Document Ref:	1309,201/FS
Project:	Ursula
Revision:	H **live**
Date:	15 Mar 1998
Author(s):	Mike Stephens
Change:	N/A

1.0 Overview

This FS covers the changes proposed in the kernel; to support enhanced memory hardware, to provide enhanced performance, and to provide enhanced features.

Public API implications are detailed in section 5. A detailed logical memory map is given in appendix C.

All details are preliminary and subject to change without notice.

5.0 Programming Interface

5.1 N/A

5.2 Video memory performance

Ursula will support a cached screen (cacheable+bufferable memory instead of uncacheable+bufferable) on Phoebe hardware. This gives significantly improved read and write bandwidths for the screen, given the unusual behaviour of SA110 for uncacheable+bufferable space. The problem of screen update delays caused by the writeback data cache is automatically handled by the kernel. The latter requires cleaning of the processor data cache ('screen cleaning') in some cases, but this is handled very efficiently by a combination of Phoebe hardware support and Ursula strategy.

In general, the presence of a cached screen is transparent to applications. Three additional reason codes for OS_ScreenMode provide for OS control and for application control where necessary. Note however, that the screen memory size is always a multiple of 1M for Ursula (even when cached screen is suspended), rather than 4k as for RISC OS 3.7.

Utilities that have been written to attempt to provide a cached screen on RISC OS 3.7 must not be used with Ursula. They are redundant, and may interfere with kernel screen support.

OS_ScreenMode 4 (SWI &65)

Controls cacheing of screen memory and screen cleaning.

When the screen is cached, cleaning of the processor data cache may be needed to avoid visual artifacts ('screen cleaning'). There are two types of cleaning; foreground cleaning, requested via OS_ScreenMode 5; background cleaning that may be initiated by the kernel on a VSync. The VSync laziness allows background screen cleaning to be often averted; eg. the kernel must clean the data cache for other reasons, and so cancels the VSync clean.

It is strongly recommended that this reason code is only used for non-desktop applications (probably only for games). Manipulation of these screen properties by a desktop application is extremely bad practice. The most likely use for games is to turn off screen cleaning, in order to avoid the small overhead of VSync cleaning by the kernel. Another possible use is to turn off the cached screen, where cacheing does not help game performance.

Games software should note that, because of problems with processor cache coherency, the cached screen is not supported with hardware scroll; any hardware scrolling will automatically suspend the cached screen until the next mode change.

On entry

R0 =	4 (reason code)		
R1 =	-1 to read current control flags, or		
	new control flags as follows:		
	bit 0	cached screen suspended for this mode if set	
	bit 1	screen cleaning suspended for this mode if set	
	bits 2 to 30	reserved (should be 0)	
	bit 31	must be 0 to set flags	
R2 =	-1	read current VSync cleaner laziness	
	1	VSync cleaner laziness 1 (clean on first available VSync)	
	2	VSync cleaner laziness 2 (clean on second VSync)	
	3	VSync cleaner laziness 3 (clean on third VSync)	
	other values undefined		

On exit

R1 =	current control flags (either read or newly set)
R2 =	current VSync laziness (either read or newly set)

Suspension of cached screen or screen cleaning are for the duration of the current mode only.

If the platform does not support a cached screen, the current control flags and current VSync laziness returned on exit will be restricted accordingly (but no error occurs).

R1 bit 1 on entry controls suspension of both VSync cleaning and foreground cleaning via OS_ScreenMode 5; forced cleaning through OS_ScreenMode 6 is unaffected.

OS_ScreenMode 5 (SWI &65)

Request a foreground screen clean.

If cleaning is not suspended via OS_ScreenMode 4, and a clean is needed, cleans the screen immediately and cancels any pending VSync clean. Otherwise, does nothing.

Applications are unlikely to need to use this reason code. The Ursula Window Manager automatically deals with screen cleaning at the end of standard redraw transactions.

On entry

R0 = 5 (reason code)

OS_ScreenMode 6 (SWI &65)

Force a screen clean, if needed.

If a clean is needed, cleans the screen immediately and cancels any pending VSync clean. Otherwise, does nothing. The clean happens even if cleaning is suspended via OS_ScreenMode 4.

Non-desktop applications such as games may find this reason code useful. Typically, they would suspend screen cleaning via OS_ScreenMode 4 to stop kernel background cleaning on VSyncs, while retaining the performance of a cached screen. They can then use OS_ScreenMode 6 to clean the screen as necessary.

On entry

R0 = 6 (reason code)

5.3 Slow memory in the free pool

On Phoebe hardware, VRAM is slower than SDRAM when used as ordinary (non-screen) memory. The Ursula kernel automatically uses SDRAM in preference to non-screen VRAM when taking memory from the free pool, and swaps any used non-screen VRAM for SDRAM when the latter becomes free.

This is largely transparent to all software. One issue is that Wimp_ClaimFreeMemory (already deprecated since RISC OS 3.5) is further deprecated because it suspends any pending swaps of non-screen VRAM and SDRAM. Closely related to this is that old versions of the WindowManager *must not* be loaded onto Ursula; they will not properly inform the kernel of free memory claims.

Another possible minor issue is that benchmark timings (software performance) may be variable in certain circumstances. The circumstances are mainly limited to short periods after the following sequence: free memory drops below the level at which non-screen VRAM must be used, free memory then grows such that less non-screen VRAM should be in use. Hence, the circumstances are likely to be rare.

5.4 N/A

5.5 Task swapping

The Ursula kernel introduces *Lazy* task swapping. This gives much more efficient task swapping than RISC OS 3.7 (which in turn is much more efficient than the Wimp based task swapping in 3.6), particularly for large Wimp slots. Lazy task swapping cannot be used on all variants of the current StrongARM core; this is dealt with automatically on kernel boot.

Lazy task swapping is largely transparent to all software. The following should be noted. Lazy task swapping means that at any given time application space may not be contiguously mapped (but will of course appear so if accessed). Lazy task swapping causes a large number of expected data and prefetch aborts, but these are silently handled by the kernel pre-veneers.

5.6 N/A

5.7 Miscellaneous performance enhancements

A large number of kernel performance enhancements have been made. They should have virtually no programming interface impact. The following should be noted.

Only a fraction of machine memory may be cleared to zero at system boot. This should not affect software, since memory in the free pool has undefined contents anyway.

Dynamic area names are now truncated to a maximum of 31 characters.

A few OS SWI error messages (those that can occur silently very often) will not be properly internationalised. This is a minor issue because they should rarely if ever be reported.

5.8 Logical memory map changes

Most memory map changes involve private areas and so have no impact on properly written applications. The following should be noted:

SVC stack size is increased from 8k to 32k, but SVC stack space should still be treated very carefully as a scarce resource (eg. module C code should beware of profligate local variable usage).

System Sprites area maximum size is reduced to 16M (from RAM limit). This should only affect old software. The reason for the change is to avoid address space exhaustion on large memory machines.

Maximum RMA size is increased from 11M to 15M; this should only have beneficial effect.

Some kernel workspace is better protected from user access; this will only affect broken software. Address zero may be protected from User mode reads as well as writes (eg. to support lazy null pointer protection); this will only affect broken software.

There is no longer a logical copy of physical DRAM space. This should not affect non-Acorn software, as the region is private.

The screen may now be cacheable, and mapped at 1M granularity rather than 4k. This should be largely transparent, but the granularity may confuse some software that attempts to resize the screen.

See appendix C for a detailed logical memory map.

5.9 Dynamic area enhancements

Ursula supports two new types of dynamic area; Shrinkable areas and Sparse areas. It also supports clamping of address space used by dynamic areas as a means to avert address space exhaustion on large memory machines. The API now defines a concept of binding dynamic areas to applications that create them. This also provides room to better use address space, but is unlikely to be fully implemented until a realease beyond Ursula.

The API changes are; extension to OS_DynamicArea reason code 0, addition of OS_DynamicArea reason codes 5 to 10.

Note that there may be some issues for software such as editors that attempt to read dynamic areas they do not own. The binding of areas to applications and the presence of Sparse areas mean that dynamic areas are no longer guaranteed to be at disjoint addresses, nor always accessible, nor always mapped simply from their base address. Software that reads a foreign area without understanding this may read garbage or cause a data abort.

OS_DynamicArea 0 (SWI &66)

Creates a dynamic area. Ursula extends this reason code to the following:

On entry:
r0 = reason code (0)
r1 = new area number, or -1 => RISC OS allocates number
(-1 is reserved for Acorn use)
r2 = initial size of area (in bytes)
r3 = base logical address of area, or -1 => RISC OS allocates address
space (-1 is reserved for Acorn use)
r4 = area flags
bits 0..3 = access privileges
bit 4 = 1 => not bufferable
bit 5 = 1 => not cacheable
bit 6 = 0 => area is singly mapped
= 1 => area is doubly mapped
bit 7 = 1 => area is not user draggable in TaskManager window
bit 8 = 1 => area may require specific physical pages
bit 9 = 1 => area is Shrinkable (new)
bit 10 = 1 => area is Sparse (new)
bit 11 = 1 => area is bound to client application (new)
bits 12..31 reserved
r5 = maximum size of area, (or -1 for non Sparse area, to use total
RAM size)
r6 -> area handler routine
r7 = workspace pointer for area handler (-1 => use base address)
r8 -> area description string (null terminated)

On exit:
r1 = given or allocated area number
r3 = given or allocated base address of area
r5 = given or allocated maximum size

Bits 9, 10 and 11 of r4 are new area flags; see below for overviews of their use.

Although not previously documented, setting area flags such that r4 bit 4 is 1 and bit 5 is 0 (area is apparently cacheable but not bufferable) has not been sensible. On Ursula, a meaning may be attached to this, but its use is reserved for Acorn. Therefore you should never request such flag settings either on Ursula or on earlier kernels.

Using a maximum size (r5) of -1 is now strongly deprecated because of the problems of address space exhaustion on large memory machines. Moreover, a maximum size of -1 has no valid meaning for Sparse areas. New code should set the smallest maximum size that is sensible for the use required. Because old code may often use -1, a clamp may be set via OS_DynamicArea 8 (see below) to limit the problem.

Overview of Shrinkable areas

Shrinkable areas are created with bit 9 of r4 set. They may be shrunk by the kernel when free memory is about to be exhausted. Shrinkable areas must have a handler routine. A new handler reason code is used for Shrinkable areas as follows:

TestShrink (Dynamic Area handler reason code 4):

On entry:
r0 = 4 (reason code)
r4 = current size of area (bytes)
r5 = page size (bytes)
r12 = pointer to workspace

On exit:
r3 = maximum amount area could shrink by, in bytes

Hence, the handler controls how much of a the area's current size may be regarded as free by the kernel, and reclaimed if necessary via a normal shrink.

Efficiency and Shrinkable areas

Note that the presence of Shrinkable areas imposes a load on the kernel whenever the true amount of free memory is requested; eg. for every Wimp_SlotSize SWI call. Therefore Shrinkable areas should be used sparingly, and only when there is good reason. Although no explicit limit is currently imposed, it is expected that only a handful of Shrinkable areas (say up to 10) need be supported efficiently by the kernel.

Overview of Sparse areas

Sparse areas are created with bit 10 of r4 set. They need not be contiguously mapped upwards from their base address. They may have memory arbitrarily distributed (with a granularity of the machine page size) within their address space, and gaps in the mapping are known as holes. One use of Sparse areas is to support the reclaiming of memory by a garbage collector, without forcing movement of used blocks.

The size of a Sparse area that is returned by OS_DynamicArea 2 is the total amount of memory currently mapped into the area, whatever the distribution.

If a Sparse area is created with a non-0 initial size (r2), the memory is allocated contiguously from the base upwards, but the area is free to behave subsequently in a sparse way (and cannot behave in a non-sparse way).

The 'maximum size' of a Sparse area (r5) is the size of the logical address space within which memory may be distributed, and is not limited to the RAM size of the machine. A requested maximum size of -1 has no meaning. Sparse areas must be created with the smallest maximum size that is sensible, to avoid exhausting logical address space.

Growing and shrinking of a Sparse area have no meaning. This means that a Sparse area must not have a handler, and cannot be manipulated with OS_ChangeDynamicArea. Moreover, a Sparse area must be created with the following properties:

- singly mapped	(r4 bit 6 clear)
- not user draggable	(r4 bit 7 set)
- no specific pages	(r4 bit 8 clear)
- not Shrinkable	(r4 bit 9 clear)

The mapping of a Sparse area is controlled through new OS_DynamicArea reason codes 9 and 10 (see below).

Efficiency and Sparse areas

Although the total logical address space assigned to a Sparse area ('maximum size') can be very large, applications should minimise the space requested for two reasons of efficiency. The first reason is that very large address space requests require significant RAM just to support memory management of the space (4k for every 4M of address space on current platforms). The second reason is that OS_DynamicArea 10 calls to unmap pages scattered over very large address space will be slow (the cost will be dominated by the size of space to check, rather than the number of pages that require unmapping).

In any case, large address space requests should be avoided where possible, just as large maximum sizes for ordinary areas should be avoided, because they will rapidly exhaust logical address space, and prevent the creation of other dynamic areas.

Overview of binding areas to applications

Bit 11 of r4 tells the kernel that the dynamic area is bound to the client application (ie. the area is only meaningful as part of the application that creates the area, and is not accessed by other software). Most areas created by typical applications can be bound to the application. It is strongly recommended that all new applications do create such areas with bit 11 set. Note that setting bit 11 is backward compatible with kernels on RISC OS 3.5, 3.6 and 3.7. Behaviour is undefined if a client that is not an application (eg. a module) creates an area with bit 11 set.

If bit 11 of r4 is set, the kernel is free (but not required) to do both of the following: remove the dynamic area when the application dies (if the application fails to do so); swap out the dynamic area with the application slot. The main advantage of binding areas to applications is that the kernel can make more efficient use of address space by overlaying bound areas.

OS_DynamicArea 1 (SWI &66)

Removes a dynamic area.

The API is unchanged, but the behaviour is slightly different when removing a Sparse area. Instead of shrinking the area (which would be meaningless), the kernel attempts to release the entire address range (with OS_DynamicArea 10) before removal. If the attempt fails, the area is not removed, but no attempt is made to restore the mapping distribution as before the removal.

OS_DynamicArea 5 (SWI &66)

Returns total free space, allowing for Shrinkable areas. This returns the sum of the current free pool size and all 'free' sizes returned by relevant Shrinkable area handlers. See discussion above concerning efficiency and Shrinkable areas.

On entry:
  r0 = reason code (5)
  r1 = number of single Shrinkable area to exclude from sum, or -1 to
    include all Shrinkable areas.

On exit:
  r2 = total amount of free space (bytes)

OS_DynamicArea 6

Reserved for Acorn use. You must not use this reason code.

OS_DynamicArea 7

Reserved for Acorn use. You must not use this reason code.

OS_DynamicArea 8

Sets clamps on maximum size for subsequently created dynamic areas.

This reason code is intended for configuration only. Because of its global effect, it should only be used by configuration utilities, and not by applications.

On entry:
r0 = 8 (reason code)
r1 = clamp on maximum size of (non-Sparse) areas created by
OS_DynamicArea 0 with r5 = -1, or 0 to read only
r2 = clamp on maximum size of (non-Sparse) areas created by
OS_DynamicArea 0 with r5 > 0, or 0 to read only
r3 = clamp on maximum size of Sparse areas created by
OS_DynamicArea 0 with r4 bit 10 set, or 0 to read only
On exit
r1 = previous clamp for OS_DynamicArea 0 with r5 = -1
r2 = previous clamp for OS_DynamicArea 0 with r5 > 0
r3 = previous clamp for OS_DynamicArea 0 with r4 bit 10 set

Specifying -1 in R1 or R2 means the respective clamp is the RAM limit of the machine (this is the kernel default). Specifiying larger than the RAM limit in R1 or R2 is equivalent to specifiying -1.

Specifying -1 for R3 is invalid (there is no concept of RAM limit for Sparse areas). The kernel default is for no explicit limit on Sparse area maximum size. This means that the effective limit is then the size of the largest fragment of logical address space free at creation time.

OS_DynamicArea 9

Ensures that a region of a Sparse area is mapped to valid memory.

On entry:
r0 = reason code (9)
r1 = area number
r2 = base of region to claim
r3 = size of region to claim

On exit:
r0-r3 preserved

The region (base to base+size-1) must be entirely within the address range of the Sparse area. An error is returned if memory could not be mapped to cover the entire region (eg. not enough free memory). There are no restrictions on the distribution of any mapped memory within the region before the call.

Note that although arbitrary alignment of base and size is allowed, the granularity of the mapping actually performed is the page size of the machine (as returned by OS_ReadMemMapInfo).

OS_DynamicArea 10

Allows a region of a Sparse area to be released as free memory

On entry:
  r0 = reason code (10)
  r1 = area number
  r2 = base of region to release
  r3 = size of region to release

On exit:
  r0-r3 preserved

The region (base to base+size-1) must be entirely within the address range of the Sparse area. There are no restrictions on the distribution of any mapped memory within the region before the call.

Note that although arbitrary alignment of base and size is allowed, memory can only be released with a granularity equal to the page size of the machine (as returned by OS_ReadMemMapInfo). Fragments below this granularity and not released are not accumulated across calls. Hence, release of sub-page sized regions is not useful.

5.10 Support for 32-bit code

Ursula gives limited support for 32-bit code (RISC OS 3.7 gives none). This is mainly to support a more efficient FPEmulator. However, a limited API is defined to support restricted use of 32-bit user mode code by third parties. The severe restrictions mean that uses are likely to be limited.

OS_EnterUSR32 (SWI &73)

Enter 32-bit user mode.

On entry
  no parameters
On exit
  registers preserved
  interrupt status unaltered

This SWI returns in 32-bit user mode. Behaviour is undefined unless this SWI is called from 26-bit user mode (ie. the normal RISC OS user mode). This SWI cannot be called from an address above 64M. This SWI does not use the normal SWI exit code, and does not check for callbacks.

Once in 32-bit user mode, all code is subject to the restrictions of the ARM 32-bit instruction set. Beware of instructions that are illegal in 32-bit mode (eg. TEQP), and of instructions that behave differently (eg. MOVS PC,LR).

Because 32-bit user mode is not the normal RISC OS user mode, there are additional restrictions:

32-bit user mode code must not call any SWIs, except SWI OS_EnterUSR26 to return to normal RISC OS user mode

32-bit user mode code can jump to addresses above 64M (code in dynamic areas), but must jump back below 64M before calling SWI OS_EnterUSR26

Although transient callbacks can occur (via interrupt returns), the current callback handler is not called because the API for callback handlers is not 32-bit aware.

If an abort occurs in 32-bit mode code, the register dump information will be as if from 26-bit mode; an abort address above 64M cannot be reported properly. Instead, you should use

SWI OS_ReadSysInfo 7 to read the 32-bit PC and PSR for the last abort.

OS_EnterUSR26 (SWI &74)

Enter 26-bit user mode.

On entry
  no parameters
On exit
  registers preserved
  interrupt status unaltered

This SWI returns in 26-bit user mode. Behaviour is undefined unless this SWI is called from 32-bit user mode. This SWI cannot be called from an address above 64M.

OS_ReadSysInfo 7 (SWI &58)

Read information for last unexpected abort (data or prefetch)

On entry
  r0 = 7 (reason code)
On exit
  r0 preserved
  r1 = 32-bit PC for last abort
  r2 = 32-bit PSR for last abort
  r3 = fault address for last abort

The fault address is the same as the 32-bit PC for a prefetch abort, but is the address that caused the abort for a data abort.

5.11 Long command lines

OS_CLI (SWI &05)

Process a supervisor command.

The check for command line length in Ursula is now for a maximum length of 1024 bytes including the terminator, rather than 256 bytes.

The kernel fully supports the longer command lines (kernel command, environment strings, expression evaluation and so on). Note however that soft keys are still limited to a maximum of approximately 256 characters.

It is extremely important that, at an absolute minimum, applications or modules that support commands fail gracefully if they encounter a command of more than 256 bytes. It is strongly recommended that all new modules and applications allow for command lines of at least 1k.

The main purpose of the change is to support longer pathnames in commands. This is driven by the availability of long filenames from Ursula FileCore. Note that the desktop environment still sees a pathname limit of approximately 200, because of the established Wimp message API. However, studies suggest that 200 is sufficient for typical long pathnames on systems with long filenames. The longer command line should allow all desktop software to properly deal with pathnames up to at least 200 (eg. Filer issues a Rename command quoting two long pathnames).

Commands issued through OS_CLI without Wimp message involvement are free to use longer pathnames up to the 1k command line total. This should help applications during command processing that does not rely on Wimp messaged pathnames. The DDEUtils support for open ended long command lines (not passed to OS_CLI) remains for long commands between DDEUtils clients.

Despite the availability of long filenames, users should be encouraged not to create directory structures with very long pathnames (beyond about 200) for ordinary desktop use. This is because, at Ursula release, not all layers of OS and application software will deal gracefully with longer pathnames.

OS_GetEnv (SWI &10)

Read environment parameters.

Note that the environment string returned via R0 is read only. You must not attempt to write to this string, although previous documentation did not make this explicit. On Ursula, attempting to write to this string in User mode will cause an abort.

5.12 PCI support

OS_Memory 12 (SWI &68)

Recommends a base page for a currently available (not locked down) region of physically contiguous RAM.

On entry

r0 bits 0..7	= 12 (reason code)
r0 bits 8..31	= 0 (reserved flags)
r1	= size of RAM region required (bytes)

On exit

r3	= page number of first page recommended for region

This call is an aid to creating dynamic areas with contiguously mapped RAM, such as for a PCI driver that wishes to set up RAM for PCI to host access. The area should be created with a pregrow handler that can pick up the base page number as returned by this call; the handler should then fill in page numbers incrementing consecutively from this base page. The call returns an error if a suitable block is not available.

The information returned may become invalid if pages are mapped other than for creating the dynamic area. Therefore this call should be made immediately before creating the dynamic area. The area is not expected to grow from its initial size (the size passed to OS_Memory 12); an attempt to do this with further increments of the page number may fail because the pages requested do not exist or are locked down.

Creating contiguously mapped areas places a significant load on system page disposition. Such areas should only be created where necessary. It is expected that use will be restricted to a few PCI drivers.

5.13 Reading OS identifier

OS_Byte 129 (SWI &06)

When used to read the OS version identifier.

On entry:
r0 = 129
r1 = 0
r2 = &FF

On exit:
r1 = &A7 for RISC OS 3.7, &A8 for Ursula
r2 = 0

5.14 Module service calls

The Ursula kernel supports a slightly revised module format in order to allow a much more efficient kernel implementation of service call distribution. The revised format is designed to give explicit information on service calls of interest to a module, while being backward compatible with old RISC OS kernels.

Revised format for service call handler

The meaning and format of the module header is unchanged from that in the Programmer's Reference Manual (PRM 1-205).

The entry and exit conditions for the service call handler are unchanged from those in the PRM (PRM 1-210).

A table that specifies the service calls of interest to the module is added to the specification. The presence of this table is indicated by a magic instruction as the first instruction of the service handler code. The table is then referenced by an anchor word that immediately precedes the magic instruction.

Hence, the required form of the service handler is as follows (where svc is the address of the handler code, found from offset &0C of the header as usual):

address	contains	description
svc - 4	svc_tab	offset (from module start) to service table
svc	&E1A00000	magic instruction (MOV R0,R0)
svc + 4	...	handler code as recommended in PRM 1-211
...		

The anchor word at svc-4 can contain 0, meaning that no static service table is specified. This is reserved only for rare cases (see recommended use, below); a service table must be provided where possible.

The format of the table referenced by the anchor word at svc-4 is as follows:

address	contains	description
svc_tab	svc_flags	flags word
svc_tab + 4	svc_offset	offset (from module start) to handler code
svc_tab + 8	svc_1	number of first service call of interest
svc_tab + 12	svc_2	number of second service call of interest
...	...	further service call numbers of interest
svc_tab + X	0	terminator

All bits of the flags word are currently reserved and should be 0.

The svc number fields must be listed in strict ascending order of service call number.

The svc_offset should specify an entry for code that directly dispatches a service call known to be in the set of interest. That is, it can skip the code that rejects unrecognised service call numbers as outlined in PRM 1-211. (The rejection code must remain for kernels that do not use the table.)

Requirements for new modules

New modules that have no service call handler are unaffected; they specify 0 at offset &0C of the module header as usual.

All other new modules must use the revised format. The new format is fully backward compatible while allowing newer kernels the chance to compile fast service call distribution chains.

Hence, all new modules must have the magic instruction (at svc) that identifies the new format, and also the table anchor word (at svc-4).

All new modules should have a a properly compiled table (at svc_tab) of all service calls that can be of interest to any of the module's instantiations at any time during the module's residence. The handler code itself should follow the form recommended in PRM 1-211, and the code entry from the table (svc_offset) should skip the pre-rejection code, as discussed above.

The requirement allows efficient use of the table by new kernels, while also being efficient on old kernels. (On old kernels it is the time for the handler code to reject many unrecognised service calls, rather the time to dispatch recognised calls, that will typically be most significant; hence the recommended form already in PRM 1-211.)

Only rare exceptions to the requirement for a table are allowed; namely, where the set of service calls is extremely large, or is unbounded in some way. The strong recommendation is that only if modules would require a table of more than 1k bytes (more than approximately 250 service call numbers) should the table be omitted.

Modules that seem to require table omission should be redesigned or eliminated wherever possible. Otherwise, they should still have the magic instruction at svc, and specify an offset of 0 in the anchor word at svc-4. This ensures that old format modules can always be distinguished from new format modules.

The reason for requiring only rare omission of the table is that such non-compliant modules require costly service call distribution. Note that the Ursula kernel may choose to pass service calls to non-compliant modules only after passing to all relevant compliant modules, regardless of module instantiation order. (This only affects cases where service calls may be claimed.)

Recommendation for old modules

Old modules that have no service call handler are unaffected; they specify 0 at offset &0C of the module header as usual.

All other old modules should be updated to the new format at source where possible. The reason for this is that old format modules require costly service call distribution. Note that the Ursula kernel may choose to pass service calls to old format modules only after passing to all relevant new format modules, regardless of module instantiation order. (This only affects cases where service calls may be claimed.)

It should also be feasible to update most module binaries to the new format. One scheme is to add the service table, the anchor word and a handler stub at the end of the old module. The service handler offset in the module header should be updated to point to the handler stub. The handler stub contains the magic instruction and then a branch to the old handler code.

Hence, the binary update scheme looks like this:

item	description
Header	modified, header of module (updated service offset)
Body	old, body of module
SvcTable	new, service table
Anchor	new, anchor word (offset points to SvcTable)
&E1A00000	new, magic instruction (at new service offset)
B old_svc	new, branch to old service handler code

Backward compatibility

The new format should be 100% backward compatible. The only issue is the possibility of an old module whose first instruction of its service call handler happens to be the magic instruction. Such a module would confuse a new kernel (which would most likely reject it as broken). This is an extremely unlikely event, since the magic instruction is a NOP. A simple fix would be to use a different NOP code in such a module, but a binary update as above would be far preferable.

Forward compatibility

The Ursula kernel will support old and new format modules, but all modules should be created as or moved to new format wherever possible (see discussion above). A future release of the kernel may refuse to support old format modules.

C Ursula Logical memory map

Address	Function	Size	Type
000 00000			
	Reserved	8M	Private
FF8 00000			
	Dynamic areas	1.5G-8M	Public [5]
A00 00000			
	PCI space	256M	Private
900 00000			
	Copy of physical space	256M	Private
800 00000			
	Dynamic areas	2G-136M	Public [5]
088 00000			
	Reserved	3M	Private
085 00000			
	Reserved	1M-8k	Private
084 02000			
	UNDEF stack	8k	Private
084 00000			
	Reserved	4M	Private
080 00000			
	Dynamic areas	64M	Public [5]
040 00000			
	ROM	8M	Private
038 00000			
	Reserved for 2nd processor	1M	Private
037 00000			
	Reserved	1M	Private
036 00000			
	VIDC20	1M	Private
035 00000			
	Reserved for VIDC1 emulation	1M	Private
034 00000			
	I/O space	4M	Private [4]
030 00000			
	RMA	15M	Public [3]
021 00000			
	Reserved	544k	Private
020 78000			
	Reserved for fake screen	960k	Private
01F 88000			
	Reserved	416k	Private
01F 20000			
	Reserved	32k	Private
01F 18000			
	Reserved	32k	Private
01F 10000			
	Reserved	24k	Private
01F 0A000			
	"Nowhere" (2 pages)	8k	Private
01F 08000			
	Cursor/system/sound	32k	Private
01F 00000			
	System heap	3M-32k	Private
01C 08000			
	SVC stack	32k	Public [2]
01C 00000			
	Application space	28M-32k	Public
000 08000			
	Scratch space	16k	Public [1]
000 04000			
	System workspace	16k	Private
000 00000			

[1] to [5] Notes as in Programmer's Reference Manual 5a-39