Engineering Support Application Note
Number: 295
Issue: 2.02

Introduction to StrongARM and Programming Guidelines

This Application Note introduces the StrongARM SA-110 processor, and details the issues which arise owing to the differences between it and earlier ARM processors. We also introduce RISC OS 3.7, and describe its enhancements over RISC OS 3.6.

Applicable Hardware : Any system equipped with RISC OS 3.7

Related Application Notes: None

Engineering Support
Acorn Risc Technologies
Acorn House
645 Newmarket Road
Cambridge
CB5 8PB

Introduction to the StrongARM

The StrongARM 110 is a new high-performance processor from DigitalSemiconductor. It offers performance improvements over the ARM710 ranging from 100% to 1000%, depending on the application.

A StrongARM processor card will available for the Risc PC later this year. It will come with a new ROM set containing RISC OS 3.70, a new StrongARM-compatible version of RISC OS. These documents provide information on writing software, and modifying existing software, for StrongARM-based machines.

Significant StrongARM features

* 202MHz core clock

* 5-stage pipeline (Fetch, Issue, Execute, Buffer, Write)

* Separate 16K write-back data cache and 16K instruction cache

* 8 entry write buffer, each entry holding 1-16 bytes

* Fast 32 and 64 bit result multiply instructions

* Averages fewer cycles per instruction than previous ARMs

Incompatibilities

The StrongARM has two significant differences from previous ARMs that can affect existing programs. Firstly, the split caches mean that instructions written into memory may linger in the data cache, and not be in real memory when the instruction cache comes to fetch them. Likewise if some code is already in the instruction cache, and it is altered, the changed code will not be noticed. Thus existing self-modifying code and dynamic code creation or loading will generally not work.

RISC OS 3.7 provides a call to synchronise the instruction and data caches, but this is a slow operation on the StrongARM. Self-modifying code should therefore be avoided. The most common failures here arise from custom SWI veneers that assemble code on the stack, dynamic code loading and linking and custom code squeezing and encryption.

The other significant StrongARM change is that storing the PC to memory (using STR or STM) stores PC+8 rather than PC+12. This can generally be catered for by judicious use of NOPs to allow for both possibilities. This affects, for example, APCS stack backtraces and vector claimants (the example code on PRM page 1-107 will not work, for example).

There are other differences that are only likely to be encountered by low-level system code.

Most applications will run unmodified on StrongARM, especially those written in C and BASIC, but some code will inevitably need modification.

Guidelines for Authoring Software for StrongARM

* Code written in C or BASIC will generally work, but see issues below.

* Code using the shared C library or Toolbox is generally fine. Using RISCOS_Lib may cause problems, depending on the variant - re-linking to a revised implementation will typically be sufficient. Anything linked with ANSILib will not work. Products should _not_ be linked with ANSILib.

* Proprietary code squeezers may cause problems. Unsqueezed code or standard squeezed code with AIF headers (eg code produced with our Squeeze system) is generally fine. A new module, UnsqueezeAIF, detects squeezed images and unsqueezes them itself with a negligible performance penalty.

* An application patcher will be supplied with RISC OS 3.7 that is capable of detecting some known StrongARM-incompatible code sequences in an AIF image and replacing them before the image is executed. This is only a temporary solution though. If you find your software works only because the patcher is patching it, you should modify it so it does not need patching.

* All Absolute files must have valid AIF headers. Absolute files without AIF headers and untyped files are deprecated.

When writing in Assembler:

* Any self-modifying code or dynamically generated code is a problem. RISC OS 3.7 provides a SWI to support StrongARM synchronisation to dynamic code, but the performance penalty is typically severe; cleaning the data cache will take some time, and the entire instruction cache must be flushed. Hence, most cases of dynamic code can no longer be justified, and should be reimplemented in a static manner. One common reason for dynamic code is the calling of a SWI by SWI code passed in a register. RISC OS 3.7 provides a new SWI which implements this in an entirely static manner.

* The StrongARM pipeline is generally not a problem (even though it is now 5 levels deep rather than 3). The main issue is storing of the PC to memory using STR and STM; this now stores PC+8 instead of PC+12. Code should be made to work whether PC+8 or PC+12 is stored (so that ARM 6,7 are still supported). Use of the PC in data processing instructions (MOV, ADD etc) remains unaltered on StrongARM.

* STRB PC has an undefined result. This has sometimes been used as a shortcut for storing a non-zero semaphore value in speed critical code. This instruction should no longer be used. Use STR PC or STRB of some other register.

Other differences at the assembler level, which affect RISC OS itself but are unlikely to affect applications are:

* StrongARM does not support 26 bit configuration. It does support 26 bit mode within a 32 bit configuration.

* The control of architecture functions via coprocessor #15 is significantly different. The requirements for context switching are significantly more complex.

* The 'abort timing' (definition of values in affected registers) for data aborts has changed.

At the hardware level:

* When storing a byte, previous ARMs replicate the byte across the entire 32 bit data bus. StrongARM only outputs the byte on the relevant byte lane. The StrongARM processor card implements a compatibility fix for pre-Risc PC style podules as follows: bytes stored to word aligned addresses (byte 0) will be replicated on byte 2. This should allow most podules to work without firmware changes.

Backwards compatibility:

* Much StrongARM-ready code (especially in libraries, such as the aforementioned revised RISC_OSLib) may use the new SWI-calling mechanism. Therefore a new module "CallASWI" will be made available for use on RISC OS 3.1, 3.5 and 3.6 to provide the SWIs OS_CallASWI, OS_CallASWIR12, OS_PlatformFeatures and OS_SynchroniseCodeAreas.

Changes to Existing SWIs

OS_MMUControl (page 5a-70)

This SWI has a new reason code, to support platform independent cache and/or TLB flushing:

OS_MMUControl 1 (SWI &6B)

Cache flush request

On entry
R0 = reason code and flags
bits 0-7 = 1 (reason code)
bits 8-28 reserved (must be zero)
bit 29 set if flush of single entry, else complete flush
bit 30 set if processor TLBs are to be flushed
bit 31 set if processor caches are to be flushed
R1 = logical address, if R0 bit 29 set

On exit
R0,R1 preserved

Interrupts
Interrupt status is not altered
Fast interrupts are enabled

Processor mode
Processor is in SVC mode

Re-entrancy
SWI is not re-entrant

Use
This call implements platform independent cache and/or TLB flushing.

WARNING: This SWI reason code is only intended for occasional, unavoidable requirements for cache/TLB flushing. You should respect the performance implications of this SWI reason code, in a similar way to SWI OS_SynchroniseCodeAreas.

Currently, bit 29 is ignored, so that only whole flushing of caches and/or TLBs are supported. A cache will be cleaned before flushing, where the processor supports a write-back cache.

This reason code is not re-entrant. Interrupts are not disabled during the flush, so the cache or TLB flush can only be considered to be complete with respect to logical addresses that are not currently involved in interrupts.

OS_File 12, 14, 16 and 255 (page 2-40)

If R3 bit 31 is set on entry, the file being loaded will be treated as code, and the relevant area will be synchronised using OS_SynchroniseCodeAreas if necessary.

A filetype for Code (&F95) has been allocated, with the intention that it be a parallel of Data (&FFD), and when it is loaded a synchronise is automatically performed. However this functionality has *not* been implemented in RISC OS 3.70; you will still need to set bit 31 as described above.

Note that the *Load command calls OS_File 255 with bit 31 of R3 set, so no synchronise is required. This is to aid backwards compatibility; if no synchronisation is required, then use OS_File 255 directly.

DMA (page 5a-83)

The DMA Manager now marks pages uncacheable for both transfers from device to memory and transfers from memory to device. This is to ensure StrongARM's write-back cache is cleaned before the transfer starts.

New Service Calls

Service_UKCompression (Service call &B7)

An application that may need unsqueezing/patching has just been loaded

On entry

R0 = subreason code
- 0 -> first pass (unsqueeze)
- 1 -> second pass (patch)
- all other codes reserved

R1 = &B7 (reason code)
R2 = load address
R3 = size
R4 = execute address
R5 = filename (as passed to FileSwitch, not canonicalised)

On exit

R1 preserved to pass on, or 0 to claim if you know you have performed all required unsqueezing/patching for this pass.

R3 may be modified to indicate an altered size (eg after unsqueezing)
R4 may be modified to specify an new execute address
Other registers preserved

Use
This service call is passed around when an Absolute (&FF8) file is run.
The sequence of events is as follows.

1) The image is loaded.
2) If it does not contain an unsqueezed AIF header, then
Service_UKCompression 0 is issued.
3) Service_UKCompression 1 is issued.
4) OS_SynchroniseCodeAreas is called.

Therefore unsqueezers and patchers need not do any synchronisation except that necessary for their internal working (they may want to alter some code, synchronise, and call it before returning from the service call).

Two modules are supplied with RISC OS 3.7 that use this service call:

UnsqueezeAIF will unsqueeze squeezed AIF images on the first pass, and squeezed non-AIF images by code modification (so the unsqueeze will not occur until after stage 5 above)

AppPatcher will patch squeezed and unsqueezed AIF images containing certain common code sequences that are known to fail on StrongARM.

Scanning an application for code sequences is a relatively slow operation. Therefore a bit has been allocated in the AIF header to indicate that a program is "StrongARM-ready". If bit 31 of the 13th word of the AIF header (ie bit 31 of location &8030 in the loaded image) is set, the patching stage will be skipped (in RISC OS 3.7 by AppPatcher claiming the service call and doing nothing; in a future version of RISC OS FileSwitch may not issue the service call). The program will still be automatically unsqueezed; it is recommended that you continue to use the existing Squeeze utility provided with Acorn C/C++, and rely on the operating system to unsqueeze the image for you.

Service_ModulePreInit (Service call &B9)

A module is about to be initialised

On entry
R0 = module address
R1 = &B9 (reason code)
R2 = module length

On exit
R1 preserved to pass on

Use
This service is called just before a module is initialised. When a module is *RMLoaded:

1) The module is loaded into memory
2) The module is unsqueezed if necessary
3) Service_ModulePreInit is called
4) OS_SynchroniseCodeAreas is called

This service call is intended to allow run-time patching of modules.

New SWIs

OS_PlatformFeatures (SWI &6D)

Determine various features of the host platform

On entry
R0 = reason code (bits 0-15) and flags (bits 16-31, reason code specific)
Other registers depend upon the reason code

On exit
Registers depend upon the reason code

Interrupts
Interrupt status is unaltered
Fast interrupts are enabled

Processor mode
Processor is in SVC mode

Re-entrancy
SWI is re-entrant

Use
This new SWI is used to determine various feaures of the platform that the application or module is running on.

The particular action of OS_PlatformFeatures is given by the reason code in bits 0-15 of R0 as follows:

R0	Action
0	Read code features

OS_PlatformFeatures 0 (SWI &6D)

Read code features

On entry
R0 = 0 (reason code); all flags are reserved, so bits 16-31 must be clear

On exit

R0 = bit mask of features:

Bits	Meaning
0	Must tell OS when code areas change (by calling OS_SynchroniseCodeAreas)
1	Enabling, then immediately disabling interrupts will *not* give interrupts a chance to occur
2	Must be in 32 bit mode to read hardware vectors
3	Storing PC to memory (eg with STR or STM) stores PC+8 rather than PC+12
4	Data aborts occur with 'full early' timing (ie. as defined by ARM Architecture 4)

If bit 1 of R0 set then
R1 -> routine to call between IRQ enable & disable.

Use

This call determines features of the host processor's instruction set.

Platforms running ARM 6 or 7 cores will return with R0 bits 0-4 clear; platforms running StrongARM will return with bits 0-4 set in RISC OS 3.7 (but bit 2 should in fact be clear). For compatibility with older versions of RISC OS, you should call this SWI in the X form; if V is set on return and the error is 'SWI not known', then this can be taken as equivalent to a return with R0 bits 0-4 clear. Note that the easiest way to deal with the PC+8/PC+12 issue across all platforms is to make sure that the code is valid in either case (typically with judicious use of NOPs).

The routine pointed to by R1 is suitable for calling from any 26-bit mode; it preserves all flags and registers, and is reentrant. It should be called as follows:

```
LDR    R0, irqroutine   ; Load address previously stored in workspace
MOV    R14, PC          ; Get return address of PC+8
MOV    PC, R0           ; Call routine
...                     ; Code returns here
```

OS_SynchroniseCodeAreas (SWI &6E)

Inform the OS that code has been altered

On entry

R0 = flags
bit 0 clear Entire address space to be synchronised.
bit 0 set   Address range to be synchronised.
bits 1-31   Reserved

If R0 bit 0 is set then:
R1 = low address of range (word aligned)
R2 = high address (word aligned, *inclusive*)

On exit

R0-R2 preserved

Interrupts

Interrupt status is not altered
Fast interrupts are enabled

Processor mode
Processor is in SVC mode

Re-entrancy
SWI is not re-entrant

Use
This new SWI informs the OS that code has been newly generated or modified in memory, before any attempt is made to execute the code.

WARNING: This SWI is only intended for synchronisation with unavoidable use of dynamic code, because of the potential for large performance penalties. There is no longer any justification in RISC OS applications for frequent code modification. This call must never be used in an interrupt routine. Examples of reasonable use include one-off loading, relocation etc. of subsidiary code or libraries.

When using this SWI, you should use the ranged variant wherever possible, in order to minimise the performance penalty.

The call may take a long time to return (up to around 0.5ms), so it should not be called with interrupts disabled.

For compatibility with older versions of RISC OS, you should either have determined (with OS_PlatformFeatures) that this SWI should not be called, or always call this SWI in the X form, and ignore any error returned. On platforms that do not require code synchronisation (as indicated by OS_PlatformFeatures 0), OS_SynchroniseCodeAreas will do nothing.

Note that standard loading of applications or modules (and the standard C overlay system) are automatically handled by the OS, and do not require synchronisation. This may be defeated by custom squeezing, failure to use a standard AIF header for applications, and so on.

OS_CallASWI (SWI &6F)

Call a run-time determined SWI

On entry
R0-R9 as required for target SWI
R10 = target SWI number

On exit
R0-R9 as defined by target SWI
R10 preserved

Interrupt status
As defined by target SWI

Processor mode
As defined by target SWI

Re-entrancy
As defined by target SWI

Use

This new SWI allows a target SWI number to be determined at run time, and passed in a register. This removes the need for a common idiom of dynamic code, in language library SWI veneers for example. In an APCS-R library, OS_CallASWIR12 may be more appropriate (see below).

Note that OS_CallASWI is merely an alias for calling the target SWI. It has no entry/exit conditions of its own, except for the special use of R10. To call a target SWI with X bit set, us the X form of the SWI number in R10; there is no distinction between OS_CallASWI and XOS_CallASWI.

You cannot call OS_CallASWI or OS_CallASWIR12 via OS_CallASWI, since there is no defined final target SWI in this case.

You cannot usefully call OS_BreakPt or OS_CallAVector using OS_CallASWI, as OS_CallASWI will corrupt the processor flags before entering the target SWI.

For future compatibility, you should always use this SWI in preference to any local construction for calling a SWI by number. For compatibility with older versions of RISC OS, a new CallASWI module will be made available (see below).

Note that this SWI calling mechanism is almost certainly faster than any other alternative implementation, including the original _kernel_swi and _swix code contained in older versions of the SharedCLibrary. The new SharedCLibrary now simply uses OS_CallASWIR12 for _kernel_swi and _swix.

OS_CallASWI cannot be called from BASIC as BASIC only passes registers R0-R7 via its SYS instruction. It would not be useful anyway.

OS_AMBControl (SWI &70)

This SWI is for system use only; you must not use it in your own code.

OS_CallASWIR12 (SWI &71)

Call a run-time determined SWI

On entry

R0-R9 as required for target SWI
R12 = target SWI number

On exit

R0-R9 as defined by target SWI
R12 preserved

Interrupt status

As defined by target SWI

Processor mode

As defined by target SWI

Re-entrancy

As defined by target SWI

Use

This call is identical to OS_CallASWI, except that it uses R12 to specify the target SWI. This may be more convenient in some environments. In particular under APCS-R, R10 is the stack limit pointer, which must be preserved at all times; if a SWI called using OS_CallASWI were to abort or generate an error the run-time library would usually examine R10 and decide that it had no stack to handle the abort or error. Therefore APCS-R libraries must use OS_CallASWIR12 (R12 being a scratch register under APCS-R).

The CallASWI Module

The CallASWI module provides support under RISC OS 3.1, 3.5 and 3.6 for the following SWIs:

OS_CallASWI
OS_CallASWIR12
OS_PlatformFeatures
OS_SynchroniseCodeAreas

This will enable application programmers and library writers to use the new calls freely without any worries about backwards compatibility. There is no performance penalty for the use of these SWIs via the CallASWI module. One slight caveat is that older machines will not know the *names* of these SWIs; they would have to be called by number from BASIC.

Performance issues

The StrongARM has significantly different performance characteristics to older ARM processors. It is clocked 5 times faster than any previous ARM, and many instructions execute in fewer cycles. In particular

* B/BL take 2 cycles, rather than 3
* MOV PC,Rn and ADD PC,PC,Rn,LSL #2 etc take 2 cycles rather than 3
* LDR takes 2 cycles (from the cache) rather than 3, and will take only 1 cycle if the result is not used in the next instruction.
* STR takes 1 cycle rather than 2, if the write buffer isn't full
* MUL/MLA take 1-3 cycles rather than 2-17 cycles.
* Many instructions will in fact take only one cycle provided the result is not used in the next instruction.

For fuller information see the StrongARM Technical Reference Manual, available from Digital Semiconductor's WWW site (currently at http://www.digital.com/info/semiconductor/dsc-strongarm.html)

The StrongARM's cache and write buffer are also significantly better than previous ARMs, allowing an average fivefold speed increase, despite the unaltered system bus. Pumping large amounts of data will still be limited by the system bus, but advantage can be taken of the write buffer to interleave a large amount of processing with memory accesses. For example on StrongARM it is quicker to plot a 4bpp sprite to a 32bpp mode than to plot a 32bpp sprite to a 32bpp mode; the latter case is pure data transfer, while the former is less data transfer with interleaved (ie effectively free) processing.

The long cache lines of the ARM710 and StrongARM can impact performance. A random read or instruction fetch from a cached area will load 8 words into the cache; this can make traversal of a long linked list inefficient. It is also often worth aligning code to an 8-word boundary. In current versions of RISC OS modules are loaded at an address 16*n+4. Future versions of RISC OS will probably load modules at an address 32*n+4, so it is worth aligning your service call entries appropriately in preparation for this change.

Two significant disadvantages of StrongARM over previous processors are:

1) Burst reads are not performed from uncached areas. In particular this means that reads from the screen are slower on the StrongARM than on previous ARMs. A future version of RISC OS may address this by marking the screen cacheable before reading (eg in a block copy operation). Also, burst writes are not performed to unbuffered areas.

2) Code modification is expensive. You can modify code, but a complete SynchroniseCodeAreas can take of the order of half a millisecond (ie 100000 processor cycles) to execute, and will flush the entire instruction cache. Thus use of self-modifying code is strongly deprecated; a static alternative will almost always be faster. Synchronisation of a single word (eg modifying a hardware vector) is cheaper (of the order of 100 processor cycles) but still requires the whole instruction cache to be flushed.

Note that future processors will no doubt have different performance characteristics again; you shouldn't optimise your code too much for one particular architecture at the expense of others. However, hopefully you will now have a better idea how to get better performance from your StrongARM.

Use of FIQs

FIQ usage is now deprecated. When you modify the FIQ vector, you will need to synchronise the code. However, as FIQ routines cannot call SWIs you are unable to call OS_SynchroniseCodeAreas. For StrongARM compatibility we recommend you use the FIQ vector as follows:

```
 0000001C: LDR PC,&00000020
00000020: <address of FIQ handler>
```

and use the following code to do the synchronise manually when you write the instruction at location &1C (no synchronise is required when you alter the address at &20).

```
       .
       <alter FIQ vector>
       .
       MRC    CP15,0,R0,C0,C0           ; get processor ID
       AND    R0,R0,#&F000
       TEQ    R0,#&A000
       BNE    NotStrongARM
       MOV    R0,#&1C
       MCR    CP15,0,R0,C7,C10,1        ; clean data cache entry for FIQ vector
       MOV    R0,#0
       MCR    CP15,0,R0,C7,C10,4        ; drain write buffer
       MCR    CP15,0,R0,C7,C5           ; flush whole instruction cache
NotStrongARM
       .
       .
       .
```

If you want to write a complete FIQ handler into locations &1C to &FC, you should clean each 32-byte data cache line containing written code thus:

replace

```
        MOV     R0,#&1C
        MCR     CP15,0,R0,C7,C10,1
```

with

```
        MOV     R0,#&E0               ; clean complete FIQ area
 01     MCR     CP15,0,R0,C7,C10,1    ; 32 bytes (1 cache line) at a time
        SUBS    R0,R0,#&20
        BGE     %BT01
```

This will usually be slower than the approach recommended above.

This is, of course, not a future-proof solution. We recommend that no new products use FIQ code. If you feel you have a pressing need to use FIQs, contact ART Developer Support for advice.

Miscellaneous Changes in RISC OS 3.7 Not Directly Related to StrongARM

The Kernel
The System ROM is now marked read-only in the MMU page tables of machines with an ARM7 or later processor. Attempts to write to ROM space will cause a data abort.

The kernel is now aware of multiple applications (see below).

FileSwitch
FileSwitch now supports 2048-byte buffers (see FSEntry_Open on page 2-531).

The Window Manager
The Window Manager has now delegated application memory management to the kernel. The kernel is now aware of multiple application memory blocks (AMBs) and pages them in and out when requested by the Window Manager. This improves task switching performance, as the kernel is able to remap AMBs far faster than the Window Manager was able to using OS_SetMemMapEntries.

The Font Manager
The Font Manager now supports background blending in modes with 256 colours or more. This blending causes the anti-aliased pixel data to be blended with the background colour rather than using the fixed colour specified in the various colour setting SWIs.

To use the blending, you should set bit 11 of R2 in the call to Font_Paint. However, you need to ensure the following:

1) The font colours specify that there is >= 1 anti-aliasing colour, otherwise the Font Manager will attempt to paint from 1bpp cache data rather than the 4bpp anti-aliased data.

2) The Font Manager you are calling supports this bit, as previous versions of the Font Manager will complain if they find this bit set. You should call Font_CacheAddr to check the version of the Font Manager your application is running on, and only set the bit on Font Manager 3.35 or later.

There is a noticeable speed hit in using blending, so you should not use it if you know you are plotting onto a uniform background.

Blending should not be activated during printing.

Debugger
SWI Debugger_Disassemble is now aware of the complete ARMv4 instruction set.

Note that the LDRH, LDRSH and STRH instructions are not supported by the Risc PC memory system; although they will often work in a cached area because memory is accessed 4 or 8 words at a time, they will not work reliably. LDRSB, MULL and MLAL may be freely used (except of course they won't be backwards-compatible). Contrary to the information in the Acorn Assembler manual about the ARM7M, the long multiply instructions can be used in 26-bit modes.

Econet
A StrongARM-compatible Econet module has been placed in the System ROM, rather than the RiscPC/A7000 Econet card firmware being upgraded.

The Internet module
The Internet module supplied with RISC OS 3.70 is a major new version based on FreeBSD, a 4.4BSD-derived UNIX. It offers improved performance, and a wider API, including support for multicasting and T/TCP. All calls documented in chapter 123 of the Programmer's Reference Manual continue to work as documented, but some of the lower-level socketioctl calls, particularly those to do with route manipulation have been withdrawn. More details will be made available later.

A revised !Internet application is supplied as part of the !Boot structure and a new application provides an easy interface for configuring the various networking components built in to RISC OS 3.7

Access
Access Plus is now in the System ROM, instead of being supplied in the System application on the hard disc.

AUN
A new subreason code of Service_InternetStatus has been added; Service_InternetStatus 1 is issued when the Net module receives a network map from a gateway station.