WHITE PAPER

ARM Cortex-M3 Processor Software Development for ARM7TDMI Processor Programmers

Joseph Yiu and Andrew Frame July 2009

Overview

Since its introduction in 2006, the ARM® Cortex™-M3 processor has been adopted by an increasing number of embedded developers. Many of these developers have been developing MCUs based on the ARM7TDMI® processor and now, realizing the additional benefits that the Cortex-M3 processor gives them, are planning to migrate their software from the ARM7TDMI processor to the Cortex-M3 processor.

This article details some of the typical modifications that may be required to port code from the ARM7TDMI processor to the Cortex-M3 processor as the architectural differences between the two processors could mean that some software designed for the ARM7TDMI processor may need to be modified or recompiled in order to execute more efficiently on, or take advantage of, some of the advanced features of the Cortex-M3 processor.

The Cortex-M3 processor design has some innovative and much improved features and capabilities when compared with the ARM7TDMI processor. These new features enable developers to easily port the code and create more optimized code at the same time.

Benefits of porting from the ARM7TDMI to Cortex-M3

The richer instruction set and improved efficiency of the Cortex-M3 Thumb instruction set architecture (ISA) technology enables more optimized program code to be generated alongside other code saving advantages including;

- Existing ARM7TDMI ARM code size will be significantly reduced as the overhead for switching between ARM state and Thumb state is removed,
- The need for exception handler wrappers in assembler is eliminated, and
- Bit field operations and I/O operations can be simplified with the new bit field manipulation instructions and bit band features.

Since the Cortex-M3 processor can finish a task quicker than an ARM7TDMI, more time can be spent in sleep mode, reducing the active power of the system and thereby enabling better system performance and energy efficiency. Alternatively the higher performance can be used to implement more advanced application features.

The switch to Cortex-M3 also makes software development easier because the Cortex-M3 processor includes a comprehensive set of debug and trace features. Not only does this increase productivity and help in making debugging easier but it can also help the software developer better optimize the application software.

Bit field manipulation instructions

The Cortex-M3 processor supports a number of bit field manipulation instructions that can improve the performance and code size when dealing with bit fields, for example, in peripheral controls and communication protocol processing.

Bit banding

The Cortex-M3 processor supports two bit band memory regions, one for SRAM and one for peripherals. The use of bit band memory accesses can simplify peripheral control processes, and can reduce SRAM usage by converting Boolean variables into bit band alias accesses with each Boolean data taking only one bit.

Interrupt handling

The Nested Vectored Interrupt Controller (NVIC) in the Cortex-M3 processor is easy to use and provides flexible interrupt control with much simpler interrupt handling code along with the added benefit of priority level control to give better interrupt handling performance.

Data processing

The Cortex-M3 processor fast single cycle multiply and new hardware divide instructions can enable further optimized ARM7TDMI processor assembly data processing routines for DSP and saturate-like functionality.

Lower power features

The Cortex-M3 processor introduces a number of low power sleep modes which are not available in the ARM7TDMI processor. Entering low power mode is done by executing Wait For Interrupt or Wait For Event (WFI/WFE) instructions directly (via intrinsic functions in compilers, the Cortex Microcontroller Software Interface Standard (CMSIS) API, or using assembly code).

Memory Protection Unit (MPU)

The MPU is a configuration option of the Cortex-M3. If the MPU is implemented then it can be used to improve the robustness of the application and accelerate the detection of errors during the application development.

Instrumentation Trace Macrocell (ITM)

The ITM module in the Cortex-M3 processor can be used to provide diagnosis (or debug) messages through the Serial Wire Viewer (SWV) using standard debug hardware. For example, a retargeted function “fputc” can be implemented to write to the ITM stimulus port registers so that when the “printf” function is executed, the message is output to the Serial Wire Viewer connection, captured by the debug hardware and subsequently displayed in the ITM Viewer window.

Idiom recognitions and intrinsic functions on C source code

The Cortex-M3 processor provides a number of special instructions to increase the performance in data processing, for example, bit field insert, saturation and endian conversion for data. Some of these instructions can be generated by the C compiler using idiom recognitions.

A few examples of Idiom recognitions on C source code are shown in the following table:

Instruction	C language code that can be recognized by RealView C compiler

BFI	/* recognised BFI r0,r1,#8,#9 */
	int bfi(int a, int b)
	{
	  return (a & ~0x1FF00) | ((b & 0x1FF) << 8);
	}

USAT	/* recognized USAT r0,#8,r0 */
	int usat(int x) {
	  return (x < 0) ? 0 : (x > 255 ? 255 : x);
	}

SSAT	/* recognized SSAT r0,#9,r0 */
	int ssat(int x) {
	  return (x < -256) ? -256 : (x > 255 ? 255 : x);
	}

REV16	/* recognized REV16 r0,r0 */
	int rev16(int x)
	{
	  return
	(((x&0xff)<<8)|((x&0xff00)>>8)|((x&0xff000000)>>8)|((x&0x00ff0000)<<8));
	}

REVSH	/* recognized REVSH r0,r0 */
	int revsh(int i)
	{
	  return ((i<<24)>>16)|((i>>8)&0xFF);
	}

The intrinsic functions provided by the compiler can also be used to generate other special instructions that cannot be generated by a compiler in normal C code.

Instruction	Examples of C intrinsic function supported by RealView C compiler
WFI	void __wfi(void); // Wait for interrupt
WFE	void __wfe(void); // Wait for event
RBIT	unsigned int __rbit(unaigned int val); // Reverse bit
REV	unsigned int __rev(unaigned int val); // Reverse byte order

Modification Scenarios

The modifications required to enable ARM7TDMI processor software to run on a Cortex-M3 processor will depend on the complexity of the ARM7TDMI software and how it was written. The remainder of this article is divided into different sections, based on the type of software that needs to be ported.

Software type	Modifications
Part A: Simple applications written mostly in C	**C code should be recompiled for the Cortex-M3 to take advantage of the more powerful instruction set with only minor modifications being needed.** **Startup code (including the vector table) needs to be simplified.** **Interrupt controller setup code needs to be updated to take advantage of the integrated interrupt controller hardware and the removal of the need to provide top level interrupt handler code.** **Interrupt handler wrapper code needs to be simplified.** **Sleep mode entry code may need to be added to implement the Cortex-M3 architected low power modes for reducing power consumption.** **Stack memory layout may need to be considered due to the reduction in number of stacks that need to be supported.**
Part B: Applications with mixed C and assembly code.	**All modifications for Part A, plus a possibility of the need to:** **Change any assembler instructions to support Thumb-2.** **Replace Software Interrupt (SWI) with Supervisor Call (SVC), and update SWI (SVC) handler.** **Adding or adjusting fault handlers.**
Part C: Complex application with embedded OS, 3rd party software libraries.	**All modifications for Parts A and B, plus :** **Changing of embedded OS and 3rd party libraries to updated versions for Cortex-M3.**

PART A: Modifications for simple cases

C code

It is recommended that all C code be recompiled so that it can be both optimized for the more powerful Cortex-M3 Thumb instruction set, and also to ensure that all obsolete state switching code (between ARM and Thumb states) is removed. The ARM7TDMI processor supports both ARM and Thumb code, and although Thumb code compiled for ARM7TDMI will work, the Cortex-M3 instruction set should enable higher levels of performance whilst maintaining the industry leading code density offered by the original ARM7TDMI Thumb instruction set.

For ARM RealView® Development Suite (RVDS) or Keil MDK-ARM Microcontroller Development Kit (MDK-ARM) users selecting the correct "--cpu" option (for RVDS) or "--device" option (for MDK-ARM) will ensure that the most optimal code will be generated. For other development tools the command line option(s) may be different.

If the C code is purely ANSI C then no further changes will be required.

For non-ANSI C code functions, for example, state changing pragma directives ("#pragma arm" and "#pragma thumb") which are required to support the two different ARM7TDMI instruction sets, no longer apply and must be removed.

If make files are used for building the program image, then the make files might need to be modified as well. For example, with the RealView compilation tools

- Linker option "--entry 0x0" becomes "--entry __main"
- C compiler option "--apcs=/interwork" should be removed
- Command "tcc" should be replaced by "armcc" with Cortex-M3 set as the CPU target
- Command option "--thumb" for "armcc" is not necessary as "--cpu cortex-m3" option make sure only Thumb instructions are used.

C code that makes use of the RVDS and MDK-ARM Inline Assembler also needs to be adjusted to implement the Embedded Assembler as the Inline Assembler does not support Thumb instructions.

Startup code

The Cortex-M3 processor requires an extremely simple start up sequence which can, if desired, be written completely in C. This start up sequence consists of a vector table with an initial value for the Main Stack Pointer (MSP) followed by a vector address for each exception, with the possibility to enter "main" or C initialization code directly from the reset vector, or via optional startup code.

This differs greatly from the startup code for the ARM7TDMI processor which contains a vector table which consists of branch instructions, followed by an initialization sequence to set up the stack memory for the various modes, before finally entering the main program or C initialization code.

Simple startup code for the ARM7TDMI processor	**Simple startup code for the Cortex-M3 processor**
Vectors B Reset_Handler B Undef_Handler B SWI_Handler B PrefetchAbort_Handler B DataAbort_Handler	Vectors IMPORT __main DCD Stack_Top ; Main SP starting value DCD __main ; Enter C startup DCD NMI_Handler DCD HardFault_Handler

```
    B    IRQ_Handler
    B    FIQ_Handler
Reset_Handler ; Setup Stack for each mode
    LDR    R0,=Stack_Top
    MSR    CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
    MOV    SP, R0
    …  ; setup stack for other modes
    IMPORT     __main
    LDR    R0, =__main  ; Enter C startup
    BX    R0
```

```
    … ; vectors for other handlers
```

Refer to Appendix A for example C code which can be used as start up code.

The startup code can also contain directives that define the instruction set. For RVDS and MDK-ARM users, the “ARM” or “CODE32” directives should be removed from the startup code, or changed to “THUMB” in all cases. For existing code with “CODE16” directive no changes are necessary. Changing “CODE16” to “THUMB” is not recommended because “CODE16” assembles according to the legacy Thumb syntax rules rather than using the newer Unified Assembly Language (UAL). For example, under the “CODE16” syntax rules, some data processing instructions without an “S” suffix will encode to flag-setting variants whereas under the UAL syntax rules with the “THUMB” directive the “S” suffix must be explicitly specified.

Assembly code type	Recommended porting method
Assembler with “ARM” or “CODE32” directive	Change the use “THUMB” directive. The instructions will be assembled as Thumb code. In the few occasions that an instruction is not supported, the ARM assembler will report an error and the modification can be carried out manually.
Assembler with “CODE16” directive	No changes are necessary.

Example startup code and project setup can usually be found in the examples provides with Cortex-M3 development tools, and on the web sites of Cortex-M3 processor microcontroller vendors.

Interrupt Controller Setup code

The Cortex-M3 processor contains an integrated Nested Vectored Interrupt Controller (NVIC), while microcontrollers based on the ARM7TDMI processor would generally have a separate interrupt controller. Since the address location and the programmer’s model for the interrupt controllers are likely to be different, the setup code for the interrupt controllers would likely be different. In the Cortex-M3 processor, the setup for each interrupt will involve:

- Enabling and disabling of the interrupt using NVIC Interrupt Set Enable Register (ISER) and Interrupt Clear Enable Register (ICER) registers
- Optionally configuring the priority level of the interrupt using NVIC Interrupt Priority Registers

In addition, the interrupt controller setup code on the Cortex-M3 processor can be extended to include programming of the NVIC Application Interrupt and Reset Control Register (AIRCR) to define the

priority grouping (PRIGROUP) field. This allows the priority field in each priority register to be divided into pre-empt priority and sub-priority, which affects the priority level arrangement.

Interrupt Wrapper code

For the ARM7TDMI processor, it is common for the interrupt handling to be implemented with an assembly code wrapper. This is required for nested interrupt handling, and for redirecting of interrupt requests to various interrupt handlers due to the IRQ vector being shared between all interrupts except FIQ (Fast Interrupt). This type of code wrapper can be completely eliminated on the Cortex-M3 processor because the exception mechanism automatically handles nested interrupts and executes the correct interrupt handler.

Cortex-M3 processor exception handlers can be programmed as a C function with the saving and restoring of registers R0-R3, R12, R13, PSR and PC being carried out by the processor as part of its exception entry and return sequence. In addition, the return address is adjusted automatically before being saved onto the stack. With the ARM7TDMI, the return address needed to be adjusted by exception handler wrapper (or was adjusted by the RealView C compiler when the "__irq" keyword is used).

Note that existing interrupt handler code with the "__irq" keyword does not need to be changed and keeping it may add clarity to the code.

Sleep mode

The Cortex-M3 processor has architectured sleep modes which can be entered using the WFI (Wait-For-Interrupt) or the WFE (Wait-For-Event) instructions. In C programming with ARM development tools, these two instructions are accessible via the __wfi() and __wfe() intrinsics. The firmware or device driver library could also provide various power management functions for accessing sleep features.

Adjustments due to execution time differences

It is possible that a program might behave differently after being ported from ARM7TDMI processor to the Cortex-M3 processor due to differences in execution speed. For example, a system that uses software to generate bit-bang operations like SPI and I2C might achieve higher speeds when running on a Cortex-M3 processor based microcontroller and, as a result, it may be necessary to adjust any software timing constant for the application firmware.

Stack memory allocation

With the ARM7TDMI processor, each operating mode has a different stack (with the exception that User and System mode share a stack). This may result in a number of different stack memory regions. With the Cortex-M3 processor, there are only two stack pointers, and unless the software requires an embedded OS or requires the separation of privileged and non-privileged levels, it is possible to use just one stack region for the whole application. This makes stack memory planning much easier and often results in the stack memory size requirement on the Cortex-M3 processor being much less. The initialization of the second stack pointer (PSP – Process Stack Pointer) can be performed by the startup code or by the OS initialization sequence.

PART B: Modifications for projects with assembler code

Projects with mixed C and assembler code may require additional modifications due to differences in the programmer's model and instruction set.

Programmer's model

Whilst the ARM7TDMI and Cortex-M3 register bank (R0 to R15) registers are similar, there are a number of modifications required due to the simplification of the Cortex-M3 programmer's model. The most noticeable areas are:

Differences	ARM7TDMI	Cortex-M3
Modes	Seven modes : Supervisor, System, User, FIQ, IRQ, Abort, Undefined	Two modes: Handler mode and Thread mode. Thread mode can be privileged or non privileged.
Program Status Registers	Current Program Status Register (CPSR), and banked Saved Program Status Register (SPSR)	xPSR : containing Application, Interrupt and Execution Program Status Registers (APSR, IPSR, and EPSR). The I and F mode bits are removed. The SPSR is removed because xPSR is saved onto the stack during exception entry.
Stack	Up to six stack regions	Up to two stack regions
States	ARM state and Thumb state	Thumb state only

The Cortex-M3 programmer's model is much simpler than the ARM7TDMI because the handler mode manages all types of exception in the same way. Also, fewer banked registers results in less initialization before entering the main application. In most simple applications, only a simple vector table is required and then the rest of the coding can be done in C. Refer to Appendix A for an example C code vector table.

Interworking code

In mixed ARM and Thumb code projects for the ARM7TDMI processor, very often there are state changing branch veneers which switch the processor between ARM and Thumb states. These state changing operations can be embedded in assembler files or can be generated by the C compiler. When porting the application to the Cortex-M3 processor, these state changing operations are no longer necessary and can be removed to reduce some code overhead. No change is actually required for C source code files, but state changing code in assembler files should be removed.

Instruction Set

All Thumb instructions on the ARM7TDMI can execute on the Cortex-M3 processor. However, the legacy SWI (Software Interrupt) instruction has been renamed to SVC (Supervisor Call), and its associated handler will require modification (See SWI and SVC section).

For ARM assembler files, some care is needed to avoid the few ARM instructions that do not have a Thumb equivalent and cannot be used on the Cortex-M3 processor. These are:

Swap (SWP) and Swap Byte (SWPB) instructions

The SWP and SWPB instructions are used for atomic memory accesses, usually for semaphores or mutex (mutual exclusive) operations. In the Cortex-M3 processor 'swap' has been replaced by load and store exclusive access instructions (LDREX and STREX). Refer to Appendix B for an example of a simple locking process for a mutex.

Load / Store instructions

The Cortex-M3 processor supports the most commonly used load and store multiple address modes. The post indexing address mode with register offset is not supported by the Cortex-M3 processor. This addressing mode is only available in ARM state in ARM7TDMI. Examples of post indexing addressing are:

```
LDR   R0, [R1], R2  ; Read word from memory address indicated by R1, then add R2 to R1

STR   R0, [R1], R2  ; Write R0 to memory address indicated by R1, then add R2 to R1
```

If these instructions have been used, the Assembler from RealView Compilation tools will report it as an error. These instructions need to be updated to separate the increment step. For examples:

```
LDR   R0, [R1]  ; Read word from memory address indicated by R1
ADD   R1, R1, R2  ; Add R2 to R1
STR   R0, [R1]  ; Write R0 to memory address indicated by R1
ADD   R1, R1, R2  ; Add R2 to R1
```

Also, if the software code to be ported was written for memory address arithmetic using ARM instructions, the memory offset value will need to be changed because the value of PC would be the address of the current instruction plus 4.

The Cortex-M3 processor supports the following load and store multiple address modes:

Addressing modes	Supported in ARM7TDMI	Supported in Cortex-M3
Increment after – address increment after each access	LDMIA/STMIA, POP (LDMIA / LDMFD with SP as base)	LDMIA/STMIA, POP (LDMIA with SP as base)
Increment before – address increment before each access	LDMIB/STMIB	-
Decrement after – address decrement after each access	LDMDA/STMDA	-
Decrement before – address decrement before each access	LDMDB/STMDB, PUSH (STMDB / STMFD with SP as base)	LDMDB/STMDB, PUSH (STMDB with SP as base)

For example, the following instructions are supported in the Cortex-M3 processor:

```
LDMIA   R0!, {R1-R4}  ; Read 4 words from memory address indicated by R0, post increment R0
STMIA   R0!, {R1,R4,R6}  ; Write 3 words to memory address indicated by R0, post increment R0
```

```
LDMDB   R0!, {R1-R4}  ; Read 4 words from memory address indicated by R0, pre-decrement R0
STMDB   R0!, {R1,R4,R6}  ; Write 3 words to memory address indicated by R0, pre-decrement R0
PUSH    {R4-R6, LR} ; Save R4, R5, R6, and LR to stack
POP     {R4-R6, PC} ; Restore R4, R5, R6, and return by loading stacked LR to PC
```

The following instructions are not supported on the Cortex-M3 processor because the Thumb instruction set in Cortex-M3 processor only supports the full descending stack model:

```
STMDA R13!, {R4-R7}; Store R4, R5, R6 and R7 to stack using empty descending stack model
LDMIB  R13!, {R4-R7}; Restore R4, R5, R6 and R7 from stack using empty descending stack model
```

Conditional execution

In the ARM instruction set, most of the instructions can be conditionally executed. With the Cortex-M3 processor, the conditional execution feature is supported by placing instructions into an IT (If-Then) instruction block with each IT instruction block having up to 4 conditional instructions. With the ARM development tools, the assembler automatically inserts IT instructions into the output image resulting in a simple software porting process whilst maintaining very efficient code density. The automatic insertion of IT instructions may not be available with some 3rd party development tools in which case the IT instructions may need to be added manually.

Move to Register from Status (MRS) and Move to Status Register (MSR) instructions

Both the ARM7TDMI processor and the Cortex-M3 processor support Move to Register from Status (MRS) and Move to Status Register (MSR) instructions but the effects of these instructions differ in execution on each processor. These instructions are used for accessing special registers and are primarily involved in initialization and exception handling code as already discussed in this document.

Special registers in ARM7TDMI	Special registers in Cortex-M3	Changes
CPSR	PSR (APSR, IPSR, EPSR)	There are no I-bit and F-bit mode bits in Cortex-M3 PSR. IPSR and EPSR are added for exception status and execution status.
SPSR	Not required	Program Status Register is saved onto stack during exception automatically
Not available	CONTROL	Operation mode changed. So mode bits no longer needed, but CONTROL is added
Not available	PRIMASK, FAULTMASK and BASEPRI	I-bit in CPSR is changed to PRIMASK. F-bit is removed (because there is no FIQ in Cortex-M3). New more efficient exception mask registers FAULTMASK and BASEPRI are introduced.

In ARM7TDMI processor assembler code accesses to the CPSR are required to switch between the different processor modes. In the simpler Cortex-M3 programmer's model no equivalent processor mode definition is required so any processor mode switching code should be removed.

Interrupt enable and disable

On the Cortex-M3 processor, the interrupt enable and disable is handled by a separate exception mask register called PRIMASK. PRIMASK is primarily used to disable interrupts or exceptions during critical code sections whereas with the ARM7TDMI the I-bit and the F-bit in the CPSR are used to enable (when cleared) and disable (when set) the IRQ and FIQ exceptions

The Cortex-M3 processor interrupt enable/disable (PRIMASK) is separated from the PSR so the restoration of PSR status at interrupt returns does not change the interrupt enable/disable behaviour. The ARM7TDMI, when SPSR is restored to CPSR during interrupt return, will overwrite the I-Bit and hence might affect the interrupt enable/disable status. Some applications written for the ARM7TDMI might rely on this behaviour to re-enable interrupts at the end of interrupt handler.

The Cortex-M3 processor optionally leaves the interrupts enabled or disabled after interrupt exit. If an interrupt handler disabled the interrupts using PRIMASK due to timing critical code, clearing the PRIMASK before interrupt exit will enable the processor to accept new interrupt requests, or it can be left set to disable further interrupts and only accept NMI and HardFault exceptions.

ARM7TDMI	Cortex-M3
IRQ_handler … MRS R0, CPSR ORR R0, #0x80 MSR CPSR, R0 ; Set I-Bit to disable IRQ … ; critical section MOVS PC, LR ; Return and restore CPSR ; I-bit and F-bit are also restored so ; IRQ is re-enabled	IRQ_handler … CPSID I; Set PRIMASK to disable IRQ … ; critical section CPSIE I; Clear PRIMASK to enable IRQ BX LR ; Return

The Cortex-M3 processor interrupt model is replaced by a new exception model which has priority levels assigned to each interrupt.

Change Processor State (CPS) instructions allow the interrupt enabling or disabling operation to be done in just one instruction.

The Cortex-M3 processor does not disable interrupts when entering an interrupt or exception handler. If the ARM7TDMI application assumed that the interrupt would be disabled when entering an exception handler, and relies on this behavior, then the exception handler must also be changed.

Operation	ARM7TDMI software (assembly and C)	Porting to Cortex-M3 software (FIQ replaced by exception priority model)
Enable IRQ	MRS R0, CPSR BIC R0, #0x80 MSR CPSR, R0 __enable_irq();	CPSIE I __enable_irq();
Disable IRQ	MRS R0, CPSR ORR R0, #0x80 MSR CPSR, R0 __disable_irq();	CPSID I __disable_irq();
Enable FIQ	MRS R0, CPSR BIC R0, #0x40 MSR CPSR, R0 __enable_fiq();	CPSIE I __enable_irq();
Disable FIQ	MRS R0, CPSR ORR R0, #0x40 MSR CPSR, R0 __disable_fiq();	CPSID I __disable_irq();

The “CPSIE f” and “CPSID f” instructions clear and set FAULTMASK instead of changing the F-bit in the CPSR. On the Cortex-M3 processor these two instructions are intended to be used by fault handlers.

“MOVS PC, LR” instruction

The ARM instruction “MOVS PC, LR” is used in ARM7TDMI processor as exception return. This must be changed to “BX LR” on the Cortex-M3 processor because the move instruction (MOV) is not a valid instruction for exception return.

Branch instructions

Thumb assembly code for ARM7TDMI can be reused directly for Cortex-M3 processor, except for instructions that attempt to switch to ARM state. For ARM assembly code, one area that might need modification is the branch instructions, because the branch ranges of the instructions are different.

Instruction	ARM code	Thumb (16-bit)	Thumb (32-bit)
B label	+/- 32MB	+/-2K	+/-16MB
BL label	+/- 32MB	NA – always 32-bit	+/-16MB (was 4MB in ARM7TDMI)
B<cond> label (Conditional branch)	+/- 32MB	-252 to +258	+/-1MB, or +/-16MB when branch instruction is used with IT instruction block

To port assembler code from ARM code to Cortex-M3 Thumb code, some of the branch and conditional branch instructions will need modification to specify that the larger ranged 32-bit version of the instruction is used. This is done by adding the ".W" suffix. For example,

```
BEQ.W label
```

However, in the rare occasions that the program size is so large that the 32-bit Thumb instruction cannot provide the sufficient branch range, the branch instruction should be replaced by an indirect branch. For example,

```
LDR  R0,=BranchTarget

BX  R0
```

Coprocessor instructions

The Cortex-M3 processor does not have a coprocessor interface and hence execution of coprocessor instructions on the Cortex-M3 processor will result in a usage fault exception.

Software Interrupt (SWI) update to Supervisor Call (SVC)

The SWI instruction in the ARM7TDMI processor equates to the SVC instruction in the Cortex-M3 processor. The binary encoding of the instruction remains unchanged, but the handler code needs modification if it needs to extract the immediate value from the SVC/SWI instruction.

Use of SWI instructions in application code is usually found in applications with an embedded OS. During porting of ARM7TDMI applications to the Cortex-M3 processor, the assembly wrapper code needed to extract the immediate parameter is now associated with the SVC instructions, and the additional parameters passed to the SVC. Refer to Appendix C for example code.

Changes in fault handlers

A number of new fault exception types and fault handling features are provided in the Cortex-M3 processor. If an application written for ARM7TDMI supports fault handling, the fault handling code must be modified before being used with the Cortex-M3 processor.

Fault types	Fault handling in ARM7TDMI	Fault exception in Cortex-M3
Undefined instruction	Undef mode	Usage fault
Instruction fetch bus error	Abort mode	Bus fault
Data access bus error	Abort mode	Bus fault
Internal errors due to illegal code, such as an attempt to switch to ARM state	Not available	Usage fault
MPU access rule violations	Abort mode	Memory Management Fault
Vector fetch bus error	Not available	Hard fault

In most applications, the fault handler might just reset the system itself or restart the offending process if it is running an operating system. It is also possible to utilize these fault handlers to carry out diagnosis checks and error reporting based on the fault type. The following table lists a number of possible ways to use these fault handlers:

Fault handler	Possible actions for handling varies faults
Usage fault	Program image checksum checking, SRAM read write checking.
Bus fault	Stack memory leak checking, clock or power management status check for peripherals.
Memory management Fault	Stack memory leak checking, system security check for tampering detection.

Beside the new exception types, the Cortex-M3 processor also includes a number of fault status registers that are not available in the ARM7TDMI. These registers help in trouble shooting because the cause of the fault can easily be identified by various error flags and enable embedded programmers to locate problems quickly.

With the Cortex-M3 processor, the data access bus faults can be divided into precise and imprecise. If the data bus fault is precise, the fault address can be located by the Bus Fault Address Register.

Another enhancement to the fault handler code is that there is no need to manually adjust the return address as is required in fault handlers for the ARM7TDMI. For faults including undefined instructions, bus error (excluding imprecise bus fault) and memory management faults, the return address stacked onto the stack frame is the address when the problem occurred. In the ARM7TDMI fault handlers, the fault handler needs to change the return address by a subtract operation before the return.

PART C: Modifications for projects with Embedded OS and 3rd party libraries

For software developed for the ARM7TDMI with an embedded OS, the embedded OS needs to be updated to a version that is compatible with the Cortex-M3 processor. In most cases, the applications will already need to be rebuilt due to changes in peripherals, memory maps and system features. Some of the C language based application code written by the software developers can still be used, but must still be re-compiled due to possible differences in header files and data structures used by the OS, and to also take advantage of the new or improved Thumb instructions. If there is any change in the OS Application Programming Interface (API), the application code may require more modifications.

A number of embedded OS are already available for the Cortex-M3 processor; the following table lists some of the more popular ones:

Embedded OS	Vendor
RTX Kernel	KEIL (www.keil.com)
FreeRTOS	FreeRTOS (www.freertos.org)
µC/*OS*-II	Micrium (www.micrium.com)
Thread-X	Express Logic (www.expresslogic.com)
eCos	eCosCentric (www.ecoscentric.com), (http://ecos.sourceware.org)
embOS	Segger (www.segger.com)
Salvo	Pumpkin, Inc. (www.pumpkininc.com)
CMX-RTX, CMX-Tiny	CMX Systems (www.cmx.com)
NicheTask	Interniche Technologies, inc. (www.nichetask.com)
Nucleus Plus	Accelerated Technology (www.mentor.com)
µVelOSity	Green Hills Software (www.ghs.com)
PowerPac	IAR Systems (www.iar.com)
µCLinux	ARM (www.linux-arm.org/LinuxKernel/LinuxM3)
RTXC	Quadros System (www.quadros.com)
OSE Epsilon RTOS	ENEA (www.enea.com)
CircleOS	Raisonance (www.stm32circle.com/projects/circleos.php)
SCIOPTA RTOS	SCIOPTA (www. sciopta.com)
SMX RTOS	Micro Digital (www.smxrtos.com)

It may be possible to reuse a third party library, if the library is provided in source form, with a simple Cortex-M3 target re-compile of the code.

Appendix A

C code vector table example.

Exception vector table in C

```
typedef void(* const ExecFuncPtr)(void) __irq;

/* Linker-generated Stack Base addresses, Two Region and One Region */
extern unsigned int Image$$ARM_LIB_STACK$$ZI$$Limit;
extern unsigned int Image$$ARM_LIB_STACKHEAP$$ZI$$Limit;

/* selects the section names, here 'arm' is being used as a
   vendor namespace. This does not change the code state */
#pragma arm section rodata="exceptions_area"

/* Exception Table, in seperate section so it can be correctly placed at 0x0 */
ExecFuncPtr exception_table[] = {
    /* Configure Initial Stack Pointer, using linker-generated symbols and stack.h */
   (ExecFuncPtr)&Image$$ARM_LIB_STACK$$ZI$$Limit,
   (ExecFuncPtr)&__main, /* Initial PC, set to entry point  */
   NMIException,
   HardFaultException,
   MemManageException,
   BusFaultException,
   UsageFaultException,
   0, 0, 0, 0,            /* Reserved */
   SVCHandler,
   DebugMonitor,
   0,                  /* Reserved */
   PendSVC,
   SysTickHandler
   /* Configurable interrupts start here...*/
};

#pragma arm section
```

For RealView compilation tools, during linking, use the “keep” option to ensure the table is retained:

```
 --keep="exceptions.o(exceptions_area)"
```

Use a scatter loading file that specifies the vector table needs to be placed at the beginning of the program image:

```
LOAD_REGION 0x00000000 0x00200000
{
 VECTORS 0x0 0xC0
 {
   ; Exception table provided by the user in exceptions.c
```

```
   exceptions.o (exceptions_area, +FIRST)
 }
…
```

Further examples can be found in the examples of RealView Development Suite installation (CortexM3\Example2 and CortexM3\Example3) and within MDK-ARM.

Appendix B

Example of a simple locking process for a mutex.

```
// Assembler function for locking in mutex
get_lock
   LDR       R0, =Lock_Variable  ; Get address of lock variable
   LDREX   R1, [R0]  ; Read current lock value
   DMB                    ; Data Memory Barrier
                          ; Making sure memory transfer completed before next one start
   CBZ       R1, try_lock
   B          get_lock  ; Lock is set by another task, try again
try_lock
   MOVS    R1, #1
   STREX   R2, R1, [R0]  ; Try lock it by writing 1, if
                         ; exclusive access failed, return status 1 and write
                         ; process is not carried out
   DMB                   ; Data Memory Barrier
   CBZ     R2, lock_done ; Return status is 0,indicate
                         ; that lock process succeed
   B      get_lock       ; locking process failed, retry
lock_done
   BX      LR  ; return
```

Appendix C

Example use of SVC.

The wrapper code below extracts the correct stack frame starting location. The C handler can then use this to extract the stacked PC location and the stacked register values.

```
// Assembler wrapper for SVC handler in embedded assembler -
// Extracting the correct stack pointer value (stack frame starting location)
__asm void SVCHandler(void)
{
TST lr, #4
MRSEQ r0, MSP
MRSNE r0, PSP
B __cpp(SVCHandler_main) ; IMPORT of SVCHandler_main symbol is handled by __cpp() keyword
}
```

In the C language SVC handler, the immediate value can be accessed with the SVC instruction (Example of SVC C handler code from ARM application note AN179):

```
void SVCHandler_main(unsigned int * svc_args)
{
unsigned int svc_number;
/*
* Stack contains:
* r0, r1, r2, r3, r12, r14, the return address and xPSR
* First argument (r0) is svc_args[0]
*/
svc_number = ((char *)svc_args[6])[-2];
switch(svc_number)
{
case SVC_00:
/* Handle SVC 00 */
break;
case SVC_01:
/* Handle SVC 01 */
break;
default:
/* Unknown SVC */
break;
}
}
```

The stack is used to transfer parameters instead of using current value of registers when entering the SVC handler. This is required because the current values of R0 to R3 and R12 could be changed if another exception occurred just before SVC handler is executed.