Migrating ARM7 Code to a Cortex-M3 MCU
By Todd Hixon, Atmel

The ARM Cortex-M3 core has enhancements to its architecture that result in increased code execution speed, lower power consumption, and easier software development. The result is a true real-time core that overcomes real-time processing limitations of the ARM7TMI core. Over time, most ARM7-based designs will be migrated to the Cortex-M3.

Although ARM has done a lot to make it easy to port legacy code from the ARM7 to the Cortex-M3 core, more remains to be done. The purpose of this two-part article is to take you step-by-step through the porting process, so you will have no excuses when your boss asks to you to port some legacy code and have it ready by last Wednesday.

One of the most helpful things ARM has done is to make sure that Cortex-M3 support has been added to every ARM tool chain, which makes code compilation a straightforward process that can be done in just a few days in most situations. In fact, the most important consideration when migrating a legacy ARM7 design to the Cortex-M3 is selecting a device with peripheral hardware that is identical to that on the ARM7 in the current design. If the programming interface is different, new peripheral drivers will be required. This effort could add days or weeks to the schedule (never a good thing).

Using an M3 with identical peripheral hardware enables the software engineer to reuse most (if not all) of his C language driver code, saving days or even weeks of learning the nuances of new peripherals often associated with developing a robust driver from point zero. Vendor-supplied header files handle any relocation of peripheral register addresses - the developer simply includes the file for the Cortex-M3 device and recompiles the code.

There are, however, several differences between the Cortex-M3 and the ARM7TDMI that engineers must address in their designs. Initially in this article, I'll explore the issues that arise in dealing with exception vector table formatting, startup code/stack configuration, RAM functions remapping, and hardware interrupt configuration. Then I will address software interrupts, fault handling, the SWP (setwatchprops) command, instruction time, assembly language, and optimizations.

The following items are a checklist of what needs to be addressed when porting code from ARM7TDMI to Cortex-M3:

1. New exception vector table format.
2. New startup code/stack configuration.
3. Remapping RAM functions
4. New hardware interrupt configuration.
5. Software interrupts
6. Fault handlers
7. The SWP command
8. Instruction timing
9. Dealing with any hand-coded assembly language.
10. Optimizations

The Exception Vector Table

The exception vector table is where the application code tells the processor core the location of software routines to handle various asynchronous events. For ARM cores, these events include Reset (triggered by a power-up or hard reset), faults and aborts due to bus errors or undefined instructions and finally interrupts triggered by either software requests or external sources such as on-chip peripherals.

For an ARM7TDMI, the exception vector table typically consists of at least six[1] branch[2] instructions in hand-coded assembly:

```
b       Reset_Handler
b       UndefInstr_Handler
b       SWI_Handler
b       PrefetchAbort_Handler
b       DataAbort_Handler
b       .               ; Reserved vector
b       IRQ_Handler
b       FIQ_Handler
```

1. The FIQ handler can simply begin at offset 0x1c instead of a branch
2. PC-relative LDR instructions can be used instead of branches for long jumps

The exception vector table on the Cortex-M3 can be defined in C as an array of pointers (see below). The first entry is the address of the stack and the remaining entries are pointers to various exception handler functions:

```
#define TOP_STACK   0x20001000

void *vector_table[] = {
    TOP_STACK,
    System_Init,
    NMI_Handler,
    HardFault_Handler,
    MemManage_Handler,
    BusFault_Handler,
    UsageFault_Handler,
    /*
     * remaining handlers (including peripheral interrupts)
     */
};
```

Processor Modes

The ARM7TDMI has seven processor modes, six of which have their own stack pointer. One of the seven modes, “User”, operates at a lower privilege level than the others. The Cortex-M3, on the other hand, has only two modes: “Thread” and “Handler”. Thread mode can operate at either an elevated privilege level or a user level and can use either the Main Stack or the Process Stack. Handler mode always operates at privilege level with the Main Stack.

Elevated privilege levels allow access to the Processor Status Registers (CPSR and SPSR on the ARM7TDMI; APSR on the Cortex-M3) and possibly restricted memory regions as dictated by an optional Memory Protection Unit (MPU).

The following table show equivalent levels between the ARM7TDMI and the Cortex-M3:

ARM7TDMI				Cortex-M3 Equivalent			
Exception	Mode	Stack	Privilege	Exception	Mode	Stack	Privilege
-	User	usr	Normal	-	Thread	Process	Normal
-	System	usr	High	-	Thread	Process	High
FIQ	FIQ	fiq	High	-	-	-	-
IRQ	IRQ	irq	High	IRQ	Handler	Main	High
Reset	Supervisor	svc	High	Reset	Thread	Main	High
Software Interrupt	Supervisor	svc	High	Software Interrupt	Handler	Main	High
Undefined Instruction	Undefined	undef	High	MemManage or Bus Fault	Handler	Main	High
Prefetch or Data Abort	Abort	abt	High	Usage Fault	Handler	Main	High

Configuring the Processor Mode Stacks

All but the simplest ARM7TDMI systems use at least two processor modes: SVC for initialization and possibly main loop code and IRQ for interrupts. Each of the used modes must have their corresponding stack pointers initialized at reset which requires assembly code:

```
Reset_Handler:
        msr     CPSR_c, #ARM_MODE_IRQ | IRQ_DISABLE | FIQ_DISABLE
        ldr     sp, =IRQ_STACK_START    ; Set IRQ stack pointer
        msr     CPSR_c, #ARM_MODE_SVC | IRQ_DISABLE | FIQ_DISABLE
        ldr     sp, =SVC_STACK_START    ; Set SVC stack pointer
        ldr     r0, =System_Init
        blx     r0                      ; Jump to C routine System_Init
```

At reset, the Cortex-M3 automatically assigns its Main Stack Pointer (MSP) to the first entry in the exception vector table (TOP_STACK in the above example) and then jumps to the routine pointed to by the second entry, System_Init. Since the MSP used by the IRQ handlers can also be used by the main code, the Cortex-M3 can run many types of applications without any assemble code to initialize stack pointers.

Nested Interrupts

The management of interrupts from on-chip and off-chip peripherals is an important feature of microcontrollers with the most important performance metrics being latency (the time between when the event occurs and when software handles it) and jitter (how much the latency varies from event to event). Several features of the Cortex-M3 improve both latency and jitter compared to the ARM7TDMI as well as greatly simplifying the software needed to handle the interrupts.

Since the ARM7TDMI only has only two general purpose exception inputs, the IRQ and the FIQ, most SoC vendors include an interrupt controller to multiplex the multitude of interrupt sources down to a single IRQ or FIQ assertion. The IRQ/FIQ exception handler then must determine which interrupt source to process and call the appropriate software

routine. Another interrupt cannot be serviced until the routine completes and returns to the interrupted code, often making latency and jitter unacceptable for certain events. The obvious solution is to prioritize the events and allow those of higher priority to preempt those of lower priority. Implementing this on the ARM7TDMI involves an ISR "wrapper" in assembly code (see example below) that saves the processor status on the stack, changes the processor mode from IRQ back to SVC, then re-enables the IRQ and finally calls the event handler. When the handler returns, the saved mode is restored. Because the handler is called when the processor is in SVC mode, the IRQ can be asserted again by a higher priority interrupt event.

```
IRQ_Handler:
        sub     lr, lr, #4              ; Adjust and save LR_irq on IRQ stack
        stmfd   sp!, {lr}
        mrs     r14, SPSR       ; Save the old processor status and R0 on IRQ stack
        stmfd   sp!, {r0, r14}
        ldr     r14, =INTERRUPT_CONTROLLER_VECTOR_REGISTER
        ldr     r0, [r14]               ; Get vector to interrupt event handler
        msr     CPSR_c, #ARM_MODE_SVC   ; Switch to SVC Mode and enable interrupts
        stmfd   sp!, {r1-r3, r12, lr}   ; Save registers on SVC stack

        mov     lr, pc
        bx      r0                      ; Branch to the source handler

        ldmia   sp!, {r1-r3, r12, lr}   ; Restore registers from SVC stack
                                ; Switch to back IRQ mode and disable interrupts
        msr     CPSR_c, #ARM_MODE_IRQ | IRQ_DISABLE
        ldmia   sp!, {r0, r14}          ; Restore R0 and SPSR_irq from IRQ stack
        msr     SPSR_cxsf, r14
        ldmia   sp!, {pc}^              ; Return from interrupt
```

With the Cortex-M3, this wrapper code is no longer required because of the inclusion of the Nested Vectored Interrupt Controller (NVIC). The NVIC is similar to the interrupt controllers that SoC vendors typically include with an ARM7TDMI device however since it is integrated with the processor core, the NVIC can perform more sophisticated actions. The ARM7TDMI ISR wrapper code is now essentially done in hardware!

When an interrupt occurs that is a higher priority than is currently executing, the NVIC will automatically save the registers required for a call to a function compliant with the ARM Architecture Procedure Call Standard (AAPCS) and restore the registers when the function completes. Chances are the C compiler uses AAPCS which means that the NVIC can call C functions directly. Therefore, the `vector_table[]` array previously listed can contain pointers to C functions.

Configuring Interrupts

The NVIC determines which exception to handle based on the source's priority designation. The Reset, NMI and Hard Fault exceptions are fixed at the first, second and third highest priorities. The remaining exception sources have user configurable priority levels specified by their *Preempt Priority* and *Subpriority*. If an exception occurs with a higher Preempt Priority than what is currently executing, the handler for the new exception will be called. Otherwise, the new exception will be pended until all higher priority exceptions have completed. If multiple exceptions are pended within a particular Preempt Priority level, the NVIC will handle them in order of their Subpriority. Sources

with the same Subpriority level will be handled in the order of their NVIC source number.

Note that with the Preempt Priority and Subpriority fields, smaller numbers represent higher priority with '0' being the highest.

This example IRQ priority configuration will have the effects listed below:

	Preempt Priority	Subpriority
IRQ0	1	0
IRQ1	2	0
IRQ2	2	1

* IRQ0 can preempt IRQ1 or IRQ2.
* IRQ1 and IRQ2 will be pended if they occur while IRQ0 is executing.
* If both IRQ1 and IRQ2 were pended, IRQ1 will execute before IRQ2 when IRQ0 returns because IRQ1 has a higher Subpriority than IRQ2.
* If IRQ1 occurs while IRQ2 is executing, IRQ1 will not preempt IRQ2. Instead, IRQ1 will be pended and execute when IRQ2 completes.

The table below shows how the bits of the Priority Level is split between the Preempt Priority ("Pre" column) and the Subpriority ("Sub" column) based on the particular SoC Priority Level Register size (columns ranging 3 to 8) and the Priority Group setting (rows ranging 0 to 7).

Priority Level Register Size:	3		4		5		6		7		8	
Priority Group Setting	Pre	Sub	Pre	Sub	Pre	Sub	Pre	Sub	Pre	Sub	Pre	Sub
0	[7:5]	-	[7:4]	-	[7:3]	-	[7:2]	-	[7:1]	-	[7:1]	[0]
1	[7:5]	-	[7:4]	-	[7:3]	-	[7:2]	-	[7:2]	[1]	[7:2]	[1:0]
2	[7:5]	-	[7:4]	-	[7:3]	-	[7:3]	[2]	[7:3]	[2:1]	[7:3]	[2:0]
3	[7:5]	-	[7:4]	-	[7:4]	[3]	[7:4]	[3:2]	[7:4]	[3:1]	[7:4]	[3:0]
4	[7:5]	-	[7:5]	[4]	[7:5]	[4:3]	[7:5]	[4:2]	[7:5]	[4:1]	[7:5]	[4:0]
5	[7:6]	[5]	[7:6]	[5:4]	[7:6]	[5:3]	[7:6]	[5:2]	[7:6]	[5:1]	[7:6]	[5:0]
6	[7]	[6:5]	[7]	[6:4]	[7]	[6:3]	[7]	[6:2]	[7]	[6:1]	[7]	[6:0]
7	-	[7:5]	-	[7:4]	-	[7:3]	-	[7:2]	-	[7:1]	-	[7:0]

For example, if the Priority Level Register size is four bits, a Priority Group setting of 4 will cause bits [7:5] to be used as the Preempt Priority level and bit [4] to be used as the Subpriority. The remaining bits [3:0] are unused. In this case, an exception with priority 0x20 will preempt one with 0x40 (lower value is higher priority). If exceptions with

priorities 0x40 and 0x50 occur while exception 0x20 is being serviced, they will be pended (as described earlier) and the 0x40 exception will run before the 0x50 exception because the former has a higher Subpriority (bit 4 is 0 in 0x40 and is 1 in 0x50; 0 is higher priority than 1).

If it isn't exactly clear how the levels should be partitioned for a particular application, a reasonable starting point is to select Priority Group '0' which will make all of the levels preemptive (an 8-bit group register will only have 128 preemptive levels).

The following code sets the Priority Group in the NVIC:

```
#define NVIC_AIRCR  (*((unsigned int *)0xE000ED0C))

NVIC_AIRCR = (0x05fa << 16) |   /* Access key */
             (0 << 8);          /* Priority Group 0 */
```

All of the system exceptions except for the first three (Reset, NMI and Hard Fault) have user configurable priority levels.

```
#define CM3_SHPR  ((unsigned char *)0xE000ED18)

NVIC_ICPR[num - 3] = priority;
```

Peripheral interrupts are configured based on their IRQ number (IRQn):

```
#define NVIC_IPR  ((unsigned char *)0xE000E400)
#define NVIC_ISER ((unsigned int *)0xE000E100)
#define NVIC_ICER ((unsigned int *)0xE000E180)
#define NVIC_ICPR ((unsigned int *)0xE000E280)

/* Disable IRQn */
NVIC_ICER[IRQn >> 5] = 1 << (IRQn & 0x1f);
/* Set IRQn priority */
NVIC_IPR[IRQn] = priority;
/* Clear IRQn pending */
NVIC_ICPR[IRQn >> 5] = 1 << (IRQn & 0x1f);
/* Enable IRQn */
NVIC_ISER[IRQn >> 5] = 1 << (IRQn & 0x1f);
```

RAM Remap Function

ARM7TDMI SoC vendors commonly provide a mechanism in their memory controllers to select whether a non-volatile memory or a volatile memory appears at the reset vector (typically address 0x0). This allows an application to start with a fixed set of vectors in non-volatile memory and then switch to a different set of vectors later by setting up a new vector table in RAM and "remapping" the RAM to address 0x0.

With the Cortex-M3, this memory controller sleight-of-hand is no longer necessary as the NVIC allows the exception vector table to be located practically anywhere in memory. Using a new vector table is as simple as writing its offset from the beginning of its address space (either "code" or "SRAM") into the NVIC Vector Table Offset register.

```
void *new_vector_table[] = {
    0,
    new_System_Init,
    new_NMI_Handler,
    new_HardFault_Handler,
    new_MemManage_Handler,
    new_BusFault_Handler,
    new_UsageFault_Handler,
    /*
     * remaining handlers (including peripheral interrupts)
     */
};

#define SCB_VTOR  (*((unsigned int *)0xE000ED08))

unsigned int pv = (unsigned int)new_vector_table;

if (pv < 0x20000000) {
    /* vector table is in 'code' region */
    SCB_VTOR = pv;
}
else {
    /* vector table is in 'SRAM' region –
       use the offset from the beginning of SRAM
       and set bit 29 indicating that the table is in SRAM */
    SCB_VTOR = (pv – 0x20000000) | (1 << 29);
}
```

One requirement for the vector table is that it be aligned on the total number of entries rounded up to the next power of two words. For example, if the SoC vendor used 30 IRQ sources, the total number of entries including the 16 system entries would be 46. The next power of two up from 46 is 64 and the alignment for 64 4-byte words would be on a 256 byte boundary. The alignment of the vector table array can usually be specified with a toolchain-specific directive.

The FIQ

The Cortex-M3 has no direct equivalent to the ARM7TDMI FIQ interrupt. While the Cortex-M3's Non-Maskable Interrupt (NMI) may seem like a tempting substitute for the FIQ, the NMI is missing the key feature of the FIQ: the ability to preload data into shadowed registers. As the Cortex-M3 doesn't shadow general purpose registers, the most suitable FIQ replacement is just a normal IRQ with a higher priority assignment, if needed.

Executing From RAM

A common method of increasing the execution speed of critical code on ARM7TDMI devices is to execute it from internal SRAM (typically copied automatically there when tagged with a toolchain-specific "ramfunc" directive), taking advantage of the SRAM's faster access time (often zero wait-states vs. one or more wait-states with flash memory). The Cortex-M3 was designed for highest performance when executing from "code" memory (commonly internal flash) and execution from internal SRAM will be much slower since a single bus (the System bus) will be used for both instructions and data. The highest performance is realized when instructions are in the "code" memory region allowing the Cortex-M3 to perform simultaneous code and data accesses with use separate busses.

Similarly, the exception vector table should also be located in the "code" memory area. This allows the registers to be stacked to RAM at the same time that the exception vector is read.

Dealing with Hand-coded Assembly

In the 32-bit processor world, hand-coded assembly is often only used for operations that a high-level language either cannot perform directly (e.g., manipulating processor-specific registers) or is too slow. The Cortex-M3 has eliminated the need for much of the former and what remains (discussed below) can usually be encoded as inline assembly.

Code written in ARM-mode assembly for performance reasons will require either a rewrite into a high-level language or a manual translation into the Cortex-M3 Thumb/Thumb-2 instruction set. Using a high-level language is obviously easier to maintain and in many instances the compiler generates code as good as that generated by hand. However, if hand-coded assembly happens to be preferred, many ARM-mode instructions have Thumb-2 equivalents.

One ARM-mode feature often utilized is the conditional execution of an instruction to avoid the penalty of branching around it. The Cortex-M3 provides a similar capability with the "IT" (if-then) instruction which will conditionally execute the following one to four instructions based on whether a comparison is true or false. (In situations with more than four instructions conditional on a single comparison, an actual branch instruction must be used.) As an example, the following ARM assembly will clear eight bytes to the address in either register R2 or R3 depending on R1 being equal to 0 or not:

```
cmp   r1, #0
streq r0, [r2, #0]   ; if r1 is 0, write to first 4 bytes in r2
streq r0, [r2, #4]   ; if r1 is 0, write to second 4 bytes in r2
strne r0, [r3, #0]   ; if r1 is not 0, write to first 4 bytes in r3
strne r0, [r3, #4]   ; if r1 is not 0, write to second 4 bytes in r3
```

The equivalent code on the Cortex-M3:

```
cmp   r1, #0
ittee eq
streq r0, [r2, #0]   ; if r1 is 0, write to first 4 bytes in r2
streq r0, [r2, #4]   ; if r1 is 0, write to second 4 bytes in r2
strne r0, [r3, #0]   ; if r1 is not 0, write to first 4 bytes in r3
strne r0, [r3, #4]   ; if r1 is not 0, write to second 4 bytes in r3
```

This form of the IT instruction, `ittee eq`, causes the two instructions following it to execute only if the 'eq' condition is true and the two instructions after that to execute only if the 'eq' condition is false. The Cortex-M3 Technical Reference Manual has more details on the usage of the IT instruction.

Some instructions can be encoded as either 16-bit Thumb or 32-bit Thumb-2. The encoding used can be selected by adding a suffix to the instruction: ".n" for 16-bit Thumb (narrow) or ".w" for 32-bit Thumb-2 (wide). If unspecified, the assembler will typically encode for 16-bit Thumb.

Disabling Interrupts

Occasionally, an application may need to temporarily disable all processor interrupts. On the ARM7TDMI, the vendor-supplied interrupt controller may provide a global disable register or the application may set the Current Processor Status Register 'I' bit (and perhaps the 'F' bit) with the following assembly code:

```
disable_irq:
    mrs   r0, CPSR_c      ; read the CPSR
    orr   r0, r0, #0x80   ; set the I bit
    msr   CPSR_c, r0      ; write the modified CPSR

enable_irq:
    mrs   r0, CPSR_c      ; read the CPSR
    bic   r0, r0, #0x80   ; clear the I bit
    msr   CPSR_c, r0      ; write the modified CPSR
```

On the Cortex-M3, a special PRIMASK register disables all interrupts except the NMI and fault exceptions:

```
disable_irq:
    mov   r0, #1
    msr   PRIMASK, r0

enable_irq:
    mov   r0, #0
    msr   PRIMASK, r0
```

Software Interrupts

The ARM7TDMI allows software to generate an exception via the SWI instruction. This exception is typically used as an interface to system drivers or other privileged code that cannot be called directly. An example usage of the ARM-mode version of the SWI instruction is shown below:

```
    swi   0x123456
```

The ARM7TDMI responds by setting R14 to the instruction after the SWI, disabling IRQ, changing to SVC mode and jumping to the SWI exception table vector. ARM-mode SWI requests can be processed with this basic exception handler:

```
swi_exception_handler:
    stmfd sp!, {r10}
    ldr   r10, {r14, #-4}     ; get the SWI instruction
    bic   r10, r10, #ff000000 ; get the SWI operand from the instruction
        ; Code to handle event based on operand in r10
    ldmia sp!, {r10, pc}^     ; return from handler
```

The Cortex-M3 has a similar mechanism using the SVC instruction however the handler is different because the NVIC automatically stacks R0-R3, R12, LR, PC and PSR. The handler must first determine whether the Main or Process stack was used in order to access the SVC operand and any other parameters that might be passed.

```
svc_exception_handler:
    tst   lr, #4
    ite   eq
    mrseq r0, MSP
    mrsne r0, PSP
    ldr   r1, [r0, #24]       ; stacked PC
```

```
    ldrb  r1, [r1, #-2]       ; get the operand from the SVC instruction
        ; Code to handle SVC based on operand in r1
    bx    lr                  ; return from handler
```

Another difference in the two software interrupt implementations is that while an SWI exception handler is allowed to invoke another SWI exception, the Cortex-M3 NVIC cannot respond to an exception with the same priority as what is currently executing (attempting to do so will trigger a usage fault).

Fault Handlers

Both the ARM7TDMI and Cortex-M3 have special fault exceptions that it will trigger if a problem is encountered during a memory access or while processing an instruction. These faults usually indicate that either hardware or software has failed and since recovery is unlikely, a typical fault handler will simply halt after logging the state of the processor so that the problem can be addressed later.

An important item to log is the address of the instruction that was being executed when the fault occurred. This along with a disassembly of the code, the contents of the processor registers and possibly a portion of the stack frame is often enough information to pinpoint what went wrong. On the ARM7TDMI, the executing instruction can be found by subtracting four from the link register (LR) for Undefined Instruction and Prefetch Abort exceptions and by subtracting eight from the LR for Data Aborts. With the Cortex-M3, the program counter at the time of the fault is pushed onto the stack as with most exception events and can be extracted with the following code:

```
    tst   lr, #4
    ite   eq
    mrseq r0, MSP
    mrsne r0, PSP
                              ; check that r0 is a valid stack
                              ; pointer to avoid a second fault
    tst   r0, #3
    bne   skip
    ldr   r1, =STACK_ADDRESS_MIN
    cmp   r0, r1
    bmi   skip
    ldr   r1, =STACK_ADDRESS_MAX - 32
    cmp   r0, r1
    bpl   skip
    ldr   r1, [r0, #24]       ; r1 <= stacked PC
skip:
```

The Cortex-M3 has the following fault exceptions:

1. Usage fault - for undefined instructions or certain unaligned accesses.
2. Memory management fault - for attempts to access unprivileged memory.
3. Bus fault - for accessing invalid or offline memory regions.
4. Hard fault - for when the above fault exceptions cannot run.

The hard fault exception has a fixed priority level (higher than any user configurable level) and is always enabled. The other fault exceptions have a user configurable priority level and must be enabled before being used. If a fault event occurs for a disabled fault handler or if the handler has too low of a priority to run, a hard fault will be triggered. For many basic systems, only a hard fault handler is necessary to catch software errors.

The Cortex-M3 has several registers to help diagnose fault conditions:

1. Usage Fault Status Register (UFSR)
2. MemManage Fault Status Register (MMSR)
3. Bus Fault Status Register (BFSR)
4. MemManage Fault Address Register (MMAR)
5. Bus Fault Address Register (BFAR)

The three status registers (UFSR, MMSR and BFSR) can all be read as a single 32-bit word called the Combined Fault Status Register (CFSR). The MMAR and BFAR registers contain the address that caused their respective faults if the MMARVALID or BFARVALID bit is set in MMSR or BFSR. The following code reads the CFSR and the appropriate Fault Address Register:

```
ldr   r0, =0xE000ED28
ldr   r1, [r0, #0]   ; r1 <= CFSR
tst   r1, #0x80      ; MMARVALID set?
it    ne
ldrne r2, [r0, #12]  ; r2 <= MMAR
tst   r1, #0x8000    ; BFARVALID set?
it    ne
ldrne r2, [r0, #16]  ; r2 <= BFAR
```

Some bus faults may not occur until several instructions have executed after the faulting instruction (for example, an STR instruction that uses the write buffer). This case will be indicated by the IMPRECISERR bit in BFSR.

The SVC instruction cannot be used in a hard fault handler. Since the SVC exception is always a lower priority than the hard fault handler, attempts to trigger it will result in a second hard fault. The Cortex-M3 responds to a double hard fault by entering a "locked" state where only a Reset, NMI or intervention with a debugger can resume execution.

The SWP Instruction

The ARM7TDMI included a SWP instruction that provided an atomic read-then-write to a memory location. A common use of SWP is in the implementation of operating system semaphores to provide mutual exclusion between tasks.

```
take_semaphore:
    ldr   r0, =semaphore_addr
    mov   r1, #1
    swp   r2, r1, [r0]     ; Set the semaphore to 1
    cmp   r2, #1           ; Was it already set by another task?
    beq   take_semaphore   ; Yes, try again

give_semaphore:
    ldr   r0, =semaphore_addr
    mov   r1, #0
    str   r1, [r0]         ; Write a 0 to semaphore to give it back
```

The Cortex-M3 does not have the SWP instruction although the semaphore functionality can be implemented with the load exclusive (LDREX) and store exclusive (STREX) instructions.

```
take_semaphore:
    ldr   r0, =semaphore_addr
    ldrex r1, [r0]
    cbnz  r1, take_semaphore   ; Another task has the semaphore
                                       ; Try again
    mov   r1, #1
    strex r2, r1, [r0]         ; Try setting the semaphore to 1
    cbnz  r2, take_semaphore   ; Another task set the semaphore
                                       ; Try again


give_semaphore:
    ldr   r0, =semaphore_addr
    mov   r1, #0
    str   r1, [r0]             ; Clear the semaphore to 1
```

Instruction Timing

The Cortex-M3 will pipeline LDR and STR instructions when possible allowing subsequent instructions to begin executing before the previous one completes. This behavior is normally desirable as it increases overall execution speed, however it can also potentially cause any assembly code tuned for a precise timing to be off.
For example, take the case of creating a pulse on a PIO pin by writing to the SoC peripheral registers associated with setting a pin high and low:

```
ldr   r0, =pio_set_reg
ldr   r1, =pio_clear_reg
mov   r2, #1
str   r2, [r0]   ; set pin high
str   r2, [r1]   ; set pin low
```

On the Cortex-M3, this code creates a pulse whose width is two system clocks long (the execution time of the second STR instruction). A reasonable assumption would be that the addition of a NOP instruction between the two STR instructions would make the pulse one clock period longer but the pulse remains only two clocks wide because the NOP is actually executed during the second cycle of the STR instruction. Adding a second NOP instruction will lengthen the pin pulse by one clock period.

Optimizations

After the initial port is complete and the application is functioning, it makes sense to investigate the new features that the Cortex-M3 has to offer and how they might help increase the performance of the application.

The Bit Band

Past ARM instruction sets only provide accesses to memory in units of bytes (8-bit), half-words (16-bit) or words (32-bit). Modifying individual bits in memory requires three steps:

1. Reading unit of memory into a general register,
2. Perform logical operations on the register to manipulate the desired bits, and
3. Writing the unit of memory back out.

One major drawback to this method is that it is not atomic. If the thread is interrupted by an ISR that writes to the same memory unit, the memory will be corrupted by the resumed thread. To make this operation atomic, interrupts would need to be disabled

before the memory read and then re-enabled after the memory write. That's at least five operations to atomically set or clear a single bit.

The Cortex-M3 provides a mechanism to modify individual bits in memory in a simple and atomic way. Basically, a single word access within a special 32MB portion of the SRAM and Peripheral address regions is handled as an individual bit access to a word in the first 1MB of the region. For example, writing a '1' to address 0x22000000 will set bit 0 of the word at 0x20000000.

```
static unsigned int x;
unsigned int *p = (unsigned int *)(((unsigned int)&x & 0xf0000000) +
                  0x02000000 + (((unsigned int)&x & 0x000ffffc) * 32));
x = 0;
p[0] = 1;             /* set bit 0 in x */
p[1] = 1;             /* set bit 1 in x */
p[31] = 1;            /* set bit 31 in x */
if (p[0]) p[30] = 1;   /* set bit 30 in x because bit 0 is set */
/* x now equals 0xc0000003 */
```

About the author

*After receiving a BSEE from the University of Texas in 1992, **Todd Hixon** has spent most of his career developing hardware and software for various microcontroller-based products, eventually specializing in network device drivers for a major DSL modem manufacturer. He now works for Atmel where he provides specialized software solutions for Atmel's AT91 family of ARM microcontrollers.*