

There are a number of unique hardware innovations in the ARM Cortex™-M3 architecture, particular in the area of exception handling. These changes reduce interrupt latency and improve overall efficiency. We have made several modifications to the RTXC Quadros RTOS to take full advantage of these features.

### Key Exception Handling Features of Cortex-M3

ARM has made a number of enhancements to how interrupts are handled in the Cortex-M3 architecture. These include:

- Automatic state saving and restoring
- Automatic reading of the vector table entry that contains the ISR address
- Support for tail-chaining
- Dynamic reprioritization of interrupts
- Early processing of interrupts and processing of late-arriving interrupts with higher priority
- Configurable number of interrupts, from 1 to 240
- Configurable number of interrupt priorities, from 0 to 7 bits (1 to 127 levels) with sub-priorities, from 8 to 1 bits (255 to 1 interrupt sources per group)
- ISR control transfer using the calling conventions of the C/C++ standard
- Priority masking to support critical regions

For the purposes of this tech note we will focus on those that have the most impact on the RTOS.

### Nested Vectored Interrupt Controller (NVIC)

One of the more clever hardware innovations in Cortex-M3 is its high level of integration between the core and the interrupt controller. This integration allows for early processing of interrupts and processing of late-arriving interrupts with higher priority.

With the Cortex-M3 system, exception prioritization, nesting of exceptions, and saving of corruptible or volatile state is handled entirely by the core. One of the benefits of this approach is that higher priority interrupts remain enabled by the core on entry to an exception handler. Yet, the NVIC still provides the ability to explicitly block higher level interrupts using a priority mask to protect critical regions

Unlike most of the other ARM cores, the Cortex-M3 vector table contains a direct vectoring mechanism for up to 240 devices. The interrupt source decoding is handled completely by the core. Each vector location contains the address of the exception handlers rather

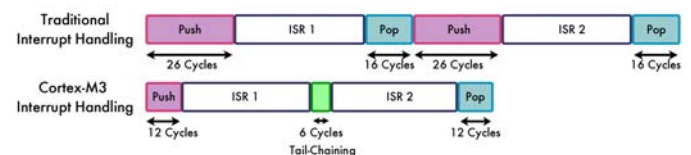
than a JMP-type instruction. The direct vector approach allows for pre-fetching of the handler code while the processor state is being saved. The core automatically saves the system state when an exception occurs and restores it on return. Since the exception handlers do not need to save or restore the system state, they can be written as ordinary ABI-compliant C functions.

### Tail chaining

The core provides a tail-chaining feature that efficiently processing back-to-back interrupts without the overhead of saving and restoring the system state. The core uses this to minimize interrupt latency and eliminate wasted cycles when dealing with pending interrupts by tail-chaining the return logic of one interrupt handler to a lower priority handler.

This tail-chaining approach is similar to a commonly used compiler optimization technique called tail-merging in which one function will share code and jump to the end of another function with similar exit logic. Tail-merging offers a reduced code footprint; tail-chaining delivers reduced interrupt latency and improved performance. The technique of tail-chaining is a cousin to tail-merging with a hardware twist.

The trick in tail chaining is to avoid the typical popping (restoring) of registers from the interrupt stack to be immediately followed by pushing (saving) the registers again for a lower priority, pending interrupt. Instead, during the interrupt return phase, the Cortex-M3 core samples the state of the interrupt controller, and if a lower interrupt is pending, then it simply begins fetching code specified by the new vector avoiding the unnecessary context store and save operation. In addition, the Cortex-M3 interrupt controller handles the prioritization and nesting of interrupts with no programmer intervention. The diagram below shows how tail-chaining can save up to 54 cycles when processing back-to-back ISRs.



### HW-based Interrupt Prolog and Epilog

It is clear that one of goals of the Cortex-M3 design team was to make it C/C++ compiler-friendly. The core automatically saves the system state when an exception occurs and restores it on return. Using their

intimate knowledge of software and hardware knowledge, ARM was able to design a system in which exception handlers do not need to save or restore the system state and thus can be written as ordinary ABI-compliant C functions. The hardware-based, core plus interrupt handling method was designed to mimic the normal volatile register-based C/C++ calling sequence as defined in the ARM EABI (Embedded Application Binary Interface). In conjunction with the tail-chaining technique described above and some special handling of register-based BX instructions, Cortex-M3 interrupt handlers simply look like high-level, asynchronous C/C++ function calls.

### **Pseudo-Device Interrupt**

A third improvement in Cortex-M3 architecture is the built-in support for a pseudo-device interrupt called PendSV (Pend Supervisor). In a typical processor and RTOS implementation, an interrupt nesting count is maintained by the RTOS. The decision whether and when to perform the task scheduling algorithm in response to an interrupt is a software-based decision based on the interrupt nesting count. In general, this decision must be performed at the conclusion of each interrupt handler.

The basic difference in Cortex-M3 is that no such software-based interrupt nesting count is required, and a dedicated interrupt handler is used to perform the RTOS scheduling logic. For example, when a device's interrupt handler performs some action, e.g., signaling a semaphore, which may result in a task context switch,

then as part of the RTOS's semaphore logic, it will trigger (or "pend") the SV device interrupt. After the device's interrupt handler completes, processor control will naturally flow to the PendSV handler which can then perform any necessary task scheduling or arbitration. The interrupt epilog of the RTOS handler implemented as a thin slice of assembly code then switches to the task's stack, restores registers, and resumes the highest priority task that is ready to run. The value proposition of the PendSV method is even more pronounced when the system must process nested interrupts at a high data rate. This significantly reduces the RTOS overhead.

### **Conclusion**

With the Cortex-M3, Arm Ltd. has designed a core which efficiently supports real-time embedded environments that require low interrupt latency and low context switch overhead combined with the ease of programming. We have adapted the RTX Quadros RTOS to take full advantage of the efficiency of these unique features.

For more details on the Cortex-M3 architecture we recommend reading the Cortex-M3 Technical Reference Manual which is available from the ARM website.

For more information on the RTX Quadros RTOS family and other products from Quadros Systems, please visit our website at [www.quadros.com](http://www.quadros.com) or talk with your local sales representative.