By Dan O'Dowd,
President,
Green Hills Software.

Real-Time Operating Systems Target Mission-Critical Embedded Systems

Today's mission-critical embedded systems require modern approaches to RTOS design. These approaches include trap mechanisms, limited re-entrancy and the use of MMUs amongst others. This article introduces Integrity, a mission-critical RTOS, and mentions points that will help all real-time programmers design systems that take full advantage of current technology.

The large software teams that are developing 100,000-line programs for today's mission-critical real-time systems are caught between a rock and a hard place. On one hand, they need a high-performance operating system that delivers fast, predictable real-time response. But at the same time, they need a secure real-time operating system that won't crash in the field.

The trouble with most real-time operating systems is that they were originally developed for 8-bit micros when a tiny footprint and fast context switching were all that mattered. As a result, most fail to leverage the protection facilities offered by today's memory management units. Facilities that are indispensable to developing and executing reliable code. Facilities that are available on most major RISC and CISC processors today. Facilities that have been available on x86 processors for more than a decade.

Fast, predictable real-time response will always be a must for mission critical systems. But equally important to the large software development teams who are working on complex projects are secure development and operating environments that facilitate the production, execution, and testing of reliable code. What designers need is a real-time operating system that combines the nimble response and determinism of a real-time microkernel with the security of a fully memory-protected RTOS.

Deeply-embedded systems can also benefit from the integrity of a memory-protected RTOS. In the past, cost and performance constraints have made designers of deeply embedded systems reluctant to utilize CPUs with integrated MMUs. But dramatic gains in CPU performance, coupled with significant improvements in integration level, have rendered traditional cost and performance concerns virtually moot. At the same time, escalating application complexity has shifted software productivity and reliability to the forefront as design issues. Within five years, integrated MMUs will be standard fare on low-cost embedded processors. RTOS vendors can do their customers a big favour by making the move to a memory-protected design as soon as possible.

SELECTING AN RTOS FOR MISSION-CRITICAL APPLICATIONS

When selecting an RTOS for a mission-critical embedded system, designers should evaluate a variety of criteria, including performance, functionality, and security. Support for pre-emptive, real-time multitasking, of course, is a prerequisite. Partitioning sophisticated real-time applications as a collection of tasks makes them easier to conceptualize, prioritize, and debug. And support for priority-based pre-emption gives designers flexible control over workload scheduling and response to asynchronous events.

The default scheduling mechanism for most RTOSs is pre-emptive, priority-based multitasking, with support for multiple priority levels. With such a mechanism, tasks run until completion unless pre-empted by a higher priority task. Tasks that have the same priority can run until completion or execute in a round-robin fashion. Some RTOSs also offer variations on this theme. Integrity, for example, supports a unique scheduling mechanism that lets programmers specify guaranteed run times for critical tasks within a given priority group.

SUBROUTINE LIBRARY VS TRAP INTERFACE

Beyond baseline multitasking facilities, designers should evaluate the RTOS' underlying architecture and how it interacts with user applications. Most real-time operating systems are implemented as libraries of C-callable subroutines. The subroutine call mechanism provides an efficient means of invoking kernel services. At the same time, the library implementation provides a scalable, modular environment that enables designers to optimize speed and size by selecting only those services that they need for their target system. The user application, together with selected RTOS services and boot code, are all linked and downloaded to the target as a single image.

The RTOSs used in mission-critical systems take a different tack, utilizing a trap mechanism that maximizes security by enabling the kernel to enforce strict

access permissions and provide extensive error checking. The trap mechanism decouples user processes from kernel-mode operations, erecting a firewall that prevents errant or malicious user processes from accessing and corrupting critical kernel data structures.

To maximize reliability and security, the Integrity kernel maintains a list of objects and permissions for each user process. When a process requests an RTOS service or access to a kernel object, it supplies a simple integer similar to a Unix descriptor, which the RTOS translates into a physical address for that object. This translation mechanism prevents errant processes from accessing kernel objects directly and crashing the kernel (i.e., by providing a wrong or uninitialized address).

KERNEL RE-ENTRANCY COMPROMISES SECURITY

One of the most frequented touted features of today's embedded real-time operating systems is kernel re-entrancy. A re-entrant kernel is one in which multiple kernel services may be invoked by multiple user processes concurrently. Re-entrancy enables kernel services to be pre-empted, which helps compensate for long kernel service run times and reduce kernel latencies.

The problem with re-entrancy is that it gives multiple kernel-mode processes concurrent access to critical kernel data structures. These structures can be protected through mutual-exclusion mechanisms such as semaphores. But these mechanisms are complex and error prone. They also degrade overall kernel performance. A better approach is to shorten the run times of kernel services, thereby minimizing kernel latency without resorting to are entrant design. Ideally, all services must complete and exit the kernel before another service runs. This ensures that multiple processes can never have concurrent access to critical kernel data structures.

PERFORMANCE CONSIDERATIONS

When evaluating RTOS performance, the two most often quoted metrics are context switching and interrupt latency. Context switching refers to the time required to pre-empt the current task, store its context, and begin processing the next task. Most RTOSs provide context switching speeds in the tens of microseconds, depending on the processor. Interrupt latency refers to the worst-case time required for a processor to execute the first line of code in an interrupt service routine (ISR) in response to a hardware interrupt. Worst-case interrupt latencies for the fastest RTOSs and processors are often less than 10 microseconds.

High-speed context switching is important for real-time applications because it minimizes the overhead associated with workload scheduling. However, this metric is often overrated, given that properly designed applications spend the bulk of their time executing user code, not switching between tasks.

More important for mission critical real-time systems is interrupt latency, which impacts the system's ability to respond quickly and predictably to asynchronous events (interrupts). Most RTOSs support a two-tier interrupt response model based on interrupt service routines (ISRs) and background RTOS tasks. ISRs are invoked automatically by hardware interrupts and execute independent of the kernel. They typically run for a brief time, performing time-critical front-line chores such as moving characters into a buffer. The bulk of the processing is handled by less critical background tasks, which are scheduled by the kernel according to priority level.

MINIMIZING WORST-CASE INTERRUPT LATENCY

A system's worst-case interrupt response time is dictated by three parameters: the time it takes the CPU to recognize a hardware interrupt and vector to the ISR; the time it takes the slowest ISR to finish executing after an interrupt has been recognized; and the longest amount of time that the kernel disables interrupts.

The speed with which a system can recognize an interrupt and vector to an ISR is typically bounded by the run time of the CPU's slowest instruction (i.e, the instruction that is being executed when the interrupt is received). Most CPUs typically disable interrupts (or elevate their priority) while executing an instruction in order to assure atomic operation. As a result, interrupts can't be recognized until the currently executed instruction is completed.

On most CPUs, multiply and divide instructions have the longest latencies. This latency is beyond the control of the real-time operating system. However, RTOS designers can help mitigate the effect of slow instruction run times by omitting multiply and divide instructions from their kernel code. Programmers who want to do the same at a system level will need to eliminate these instructions from their application code and ISRs.

The second factor that determines interrupt latency is the time that it takes the currently executing ISR to complete once an interrupt is received. Here again, it is common practice to suspend interrupts or elevate interrupt priority until the current ISR has finished executing. Unfortunately, ISR run times are also beyond the control of the RTOS. As a result, it is incumbent upon programmers to keep ISR run times short by using ISRs to perform only time-critical tasks and avoiding the use of slow instructions such as multiply and divide.

The only parameter influencing interrupt latency that is under the direct control of the kernel is the time that the kernel disables interrupts. Virtually all real-time kernels need to disable interrupts for brief periods of time in order to operate on critical kernel data structures (such as a linked list of ready-to-run tasks). RTOSs like Integrity that target mission-critical applications should never disables interrupts, leaving interrupt latency entirely in the hands of the hardware designer and application programmer.

INTERPROCESS COMMUNICATIONS

Real-time operating systems support a variety of mechanisms for supporting interprocess communications. Two of the most popular are semaphores and pipes. Semaphores provide the most efficient means of coordinating communications between processes using shared memory. However, they're also complicated and error-prone, leaving the details of establishing, accessing, and protecting shared-memory regions to the programmer.

Pipes, which leverage the protection facilities of hardware MMUs, provide a more secure paradigm for conducting interprocess communications. They're also simpler and more intuitive. To request a pipe service, processes simply designate a receiver and tell the kernel how much data they want to send. The kernel takes care of the rest, verifying access permissions, locating the necessary buffers, and performing the transfer. In this way, processes are prevented from directly accessing shared memory.

Because of their inherent simplicity and security, pipes should be used as the default interprocess communications mechanism, with semaphores engaged only for time-critical transactions. Even when developing for a CPU with no MMU, programmers should use pipes wherever possible, thereby making it easier to migrate the application to a fully-protected environment when an MMU becomes available.

MEMORY MANAGEMENT AND I/O

Most RTOSs implement the standard C/Unix memory management model, supporting fixed- and variable-sized memory block allocation from one or more memory pools on a process by process basis. To maximize security though, the RTOS should utilize the MMU to provide a firewall that affords maximum protection for memory objects that are owned by the kernel and user processes.

Integrity's memory management facilities, for example, let programmers associate any memoryobject, including physical memory and I/O registers, with user processes. The kernel, working in conjunction with the MMU, enforces access permissions for those objects, thereby preventing processes that lack the necessary permissions from deliberately or inadvertently accessing those objects.

The ability to associate physical memory and I/O register objects with user processes also lays the groundwork for a more secure approach to I/O that is ideal for slow or adequately buffered devices that don't require frequent interrupts. Typically, I/O is handled by complex, interrupt-driven device drivers that run in kernel mode. The complexity of these drivers makes them error prone, a problem that is exacerbated by the fact the drivers run in kernel mode, which gives them access to critical kernel data structures.

A simpler, more reliable approach is to substitute user processes for device drivers, mapping the buffers and necessary kernel I/O pages into the memory space of the user processes that are responsible for I/O. This enables user processes to access CPU hardware resources and perform I/O operations without having to execute in kernel mode and without having to disable interrupts.

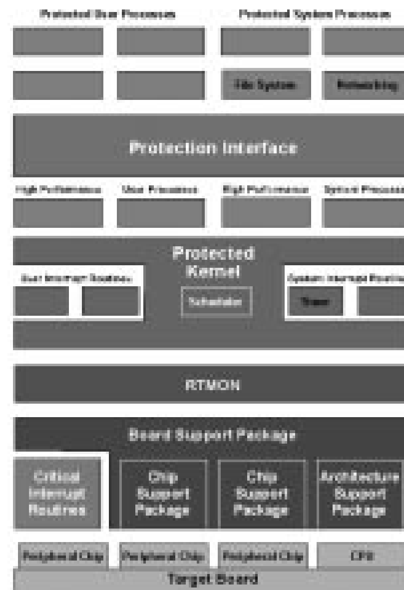

Figure 1. Integrity Block Diagram

KEEPING PACE WITH THE HARDWARE

Maximizing real-time performance and memory efficiency is still a key consideration for many real-time systems, especially deeply-embedded systems equipped with underpowered CPUs and limited memory. Even so, rapid growth in application complexity, coupled with significant gains in hardware capability, are placing security and reliability issues on an even footing with performance concerns.

Today's midrange and high-end CPU already provide the hardware MMUs needed to support advanced protection and security features. Before long, hardware MMUs will also be available on the low-cost CPUs targeting deeply embedded applications. RTOS vendors need to follow suit, adding protection and security features that can take advantage of this hardware. Designers using CPUs equipped with an MMU will be able to realize the benefits of this approach immediately. Designers using CPUs without an MMU will be ready with MMU-enabled applications that can be easily ported to more capable platforms as they become available.

Dan O'Dowd is the president of Green Hills Software, which he founded in 1982. Prior to starting Green Hills, Dan spent four years at National Semiconductor as a CPU architect and software manager, where he helped design Nationals' 32-bit NS32000 CPU, compilers, and other development tools. Dan is also a part owner of Avalon, a manufacturer of parallel processing supercomputers based on DEC's ALPHA RISC processor. Dan holds a BS in Computer Science from the California Institute of Technology.