By Linda Thompson,
Product Line Manager,
Mentor Graphics Embedded Software Division.

Enhancing your system reliability with VRTX

The engineer who develops embedded software must master a sophisticated juggling act. Today's embedded systems include more functionality than ever before, but product development schedules shrink while quality, supportability, and performance expectations continue to rise. One way to balance these competing pressures is to adopt proven commercial off-the-shelf (COTS) software where possible. But that will only result in improved quality if the software is designed and developed to appropriate standards. This article addresses governmental safety certification standards and process, such as the US FAA. It also covers selected RTOS features (preemptibility, priority inheritance, memory protection) that help ensure system reliability.

INTRODUCTION

The engineer who develops embedded software must master a sophisticated juggling act. On one hand, today's embedded systems include more functionality than ever before: sophisticated communications protocols, elaborate graphical user interfaces, internet connectivity, interprocessor communication. On the other hand, product development schedules are compressed while quality, supportability, and performance expectations increase. Heightened awareness of the importance of reliability in embedded software is evidenced in a number of areas such as:

- Public scrutiny surrounding the Y2K problem
- Pressure to avoid costly customer returns of high-volume consumer devices
- The increasing number of corporate initiatives designed to achieve 99.999% uptime in communications infrastructure equipment

One way to balance the potentially competing pressures is to adopt proven commercial off-the-shelf (COTS) software where possible. But that will only result in improved system quality if the COTS software is designed and developed to appropriate quality standards. It is as true of the real-time operating system (RTOS) kernel that will become the heart of your system as it is of your own software: reliability must first be designed in by adopting the right architecture and feature set, then built in by pursuing the right development process.

BUILDING IN RELIABILITY: THE RIGHT PROCESS

Software which will form the heart of embedded systems uch as avionics, medical devices, telecommunications equipment, and consumer devices, which require high up-time and safety levels, must be extremely reliable. Many of these types of systems are required to comply with some commercial or governmental certification process. For example an avionics system must be certified to a standard called RTCA/DO-178B (Requirements and Technical Concepts for Aviation/ DO-178B), which is jointly enforced by the U.S. Federal Aviation Administration (FAA), the Joint Airworthyness Authority, which covers the European nations, and Air Transport Canada. Other similar standards include the European Space Agency (ESA)'s PPS05, U.S. Food and Drug Administration's Good Manufacturing Practices, IEC 601-1-4 and ISO 9004.

Traditionally, vendors of COTS software for the embedded industry have not directly addressed these requirements due to the cost of developing software according to such high standards for quality and consistency of process. RTOS vendors have instead licensed their products in source form to customers requiring certification, providing a little documentation and otherwise leaving customers to their own devices for the certification effort.

However, increasing time-to-market pressures are leading many OEMs to save time and risk by selecting COTS software that complies with the standards and is certifiable out of the box. Further, even RTOS users whose applications do not require formal certification would benefit from the peace of mind that comes with knowing that the software that forms the heart of their system has been developed and validated according to the highest possible standards. For this reason, Mentor Graphics has developed its VRTXÒ RTOS kernel products to meet the requirements of certification standards such as the FAA's DO-178B.

The service history of an RTOS can play a significant role in some certifications. The fact that an RTOS has been in use for many years in a wide variety of applications with hundreds of thousands of hours of safe operation can buttress the certification effort. VRTX's service history helped VRTX achieve certification as part of the Boeing MD-11 aircraft avionics suite built by Honeywell.

RTCA/DO-178B has five levels of certification, with Level A constituting the most rigorous. The Level A standard applies to software whose failure would cause or contribute to a catastrophic failure of the aircraft. In order to address the Level A standard, the overall product development process must be controlled by written procedures that are provably followed. Requirements include:

- It must be based upon recognized industry practice

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

or approved by known experts in the industry.

- The software must be coded to recognized standards.
- Tools used to produce the software must be commercially supported and backed by a formal problem tracking system.
- A source-code control environment must track all source modifications. All modifications must be reviewed before the source is checked back in.
- All source code must be built in an independent environment.
- All object code must be directly traceable to source code and requirements

Often an automated test suite is used that exercises the software with known inputs then compares the output to desired results. But that leaves a good portion of the software untested. In real-time and event-driven systems, it may be difficult to produce the exact timing of events that would lead to certain portions of the code being executed. When Mentor GraphicsÒ XRAY Simulator's code coverage tool is used, it is not uncommon to discover that automated testing has covered as little as 50 percent to 70 percent of the code. Additional methods of testing must then supplement the automatic ones to obtain the mandated 100 percent coverage.For example, each condition statement is inspected to ensure that all the possible branches have been tested. In Level A certification, each branch must be examined using Modified Condition Branch Coverage, a process that identifies each possible Boolean value associated with a branch condition, and the software is examined for each case. A truth table is developed proving that all branches driven by the Boolean conditions function correctly. Manual examination is also typically required to meet the requirement for object-to-source code traceability, that is, to ensure that the compiler and linker have not introduced any spurious code.

DESIGNING IN RELIABILITY: THE RIGHT FEATURE SET

However, the most thorough and consistent development process cannot ensure that embedded software will perform as expected if the feature set is inadequate or the architecture is flawed. There are many kernel features which could contribute to the reliability of an RTOS-based system. One issue that has been receiving increasing attention is "fault tolerance," that is, the ability of a system to continue to operate correctly even in the presence of software faults. While a fully fault-tolerant system might include redundant hardware and a failover capability, a more limited concept of fault tolerance can be achieved by designing with a view to detection and containment of software errors. Many different RTOS features might work together to form the basis of such a capability, including the use of the processor's Memory Management Unit (MMU) to detect problems and contain errant threads.

Some implementations, based on a UNIX or POSIX process model, provide full memory protection between processes, including protecting processes' data spaces from each other. User tasks or processes are prevented from interfering with each other or with system-level operations in any way. However, the overhead created by such an implementation can be high. Data or messages which are passed between processes must be copied. In some implementations, this operation occurs during the task or process switch, which means that switch times are no longer fixed or deterministic.

An alternative approach provided in Mentor Graphics' VRTX 5 preserves performance and determinism, but still provides powerful features which can be used to detect errant threads. The MMU can be programmed to generate traps on such common programming errors as:

- Threads which under- or overflow their stacks
- Null-pointer accesses
- Attempts to execute data or overwrite code
- Accesses to unused or unmapped memory

This capability is useful during development as well as in deployed systems, as errant threads can be suspended or deleted before they adversely impact the operation of other threads in the system.

DESIGNING IN RELIABILITY: THE RIGHT ARCHITECTURE

The issues of quality and performance in a real-time environment are closely related, as the timeliness of a system's response may be as important as its accuracy. Architectural features such as deterministic system services and, in larger, more complex systems, kernel-level preemptibility and priority inheritance protocols are necessary to ensure that embedded systems interact predictably with the outside world.

If the response to an external event occurs within an ISR and if application code does not disable interrupts, the kernel's interrupt disable time, which is generally fixed and short, will be the chief impact the kernel will have on the real-time responsiveness of the system. However, in a real-world application, this is frequently not the case. The real response to an event often occurs within some task with which the ISR communicates. For example, an ISR may receive characters into a buffer and pass this buffer to a task. It is the task which processes the data and responds. In this case, the real end-to-end event response time ("interrupt task response") would include the time required to schedule the task.

A particularly interesting situation arises if an interrupt comes in while the kernel is executing a system call. (See Figure 1.) Here, a distinction must be made between preemptable and non-preemptable kernels. Most commercial RTOS kernels support preemptive task scheduling, which means that application tasks can preempt each other essentially at will. Most commercial RTOS kernels are also interruptible, which means that when an interrupt occurs while the kernel is executing, the ISR will be invoked almost immediately (within the constraints of the published interrupt latency). However, many commercial RTOS kernels are not preemptable, that is, kernel services cannot be pre-

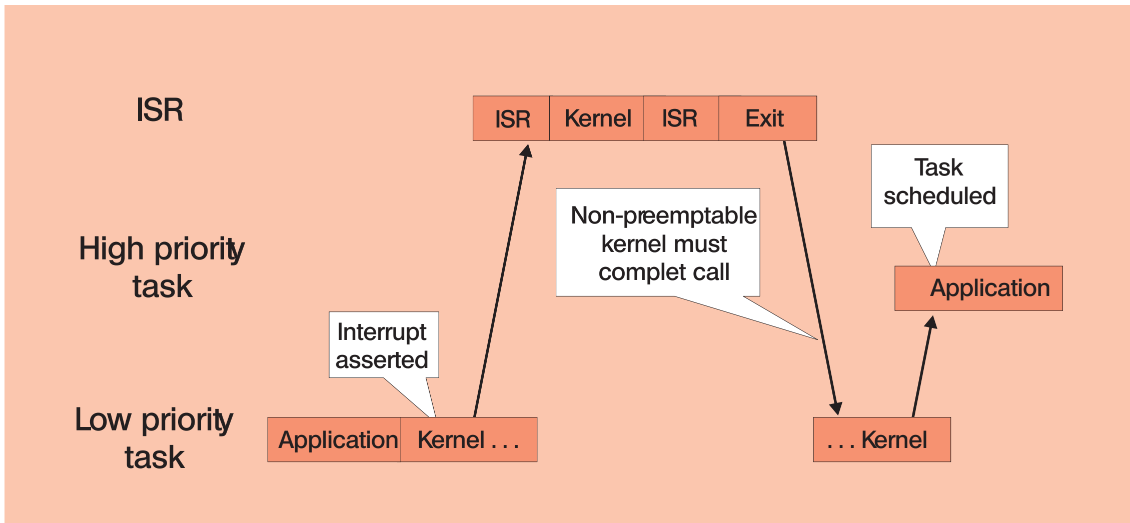

Figure 1. For a non-preemptable kernel, worst-case scheduling latency includes the time required to complete an outstanding kernel service. A preemptable kernel such as Mentor Graphics' VRTX 5 avoids this overhead.

empted. When an ISR exits through the kernel, a kernel service which was executing when the interrupt occurred must be completed before the kernel's task dispatcher can run. If the kernel was executing a non-time-critical operation on behalf of a low-priority task, this operation must complete before a high-priority task the ISR made ready can run. If a low-priority task is in the middle of a non-deterministic heap insertion operation, this scheduling latency is, in fact, unbounded. If the application is making frequent invocations of C++ constructors, such operations could be a frequent, and possibly unexpected, occurrence.

A preemptable kernel such as Mentor Graphics' VRTX 5, on the other hand, can put the low-priority operation "on hold" and dispatch the high-priority task at once. This technique minimizes scheduling latency and improves the responsiveness of the entire system.

PRIORITY INVERSION AND PRIORITY INHERITANCE

Given the obvious advantages of a preemptable architecture in minimizing scheduling latency, one is led to wonder why many popular kernels feature non-preemptable implementations. One reason is to avoid a potential problem with unbounded priority inversion.

"Priority inversion" occurs when a high-priority task blocks waiting for a low-priority task to return a semaphore. Meanwhile, any number of medium-priority tasks can preempt the low-priority task and run for an indeterminate period of time. To visualize the problem, consider the scenario illustrated in Figure 2. This scenario results in "priority inversion" because the tasks H and L behave as if their priorities were inverted. Further, the existence of one or more medium-priority tasks Mx means that the problem can persist for an unbounded period of time. This can result in unsafe and unpredictable behavior.

Priority inversion is a possible side-effect of the classic Dykstra semaphore. It can result in non-deterministic behavior, as high-priority tasks can remain blocked for indefinite periods of time waiting for medium-priority tasks to run to completion.

Unbounded priority inversion can present a problem in any application in which tasks of different priorities use traditional semaphores to share resources. It can also occur within an operating system if operating system

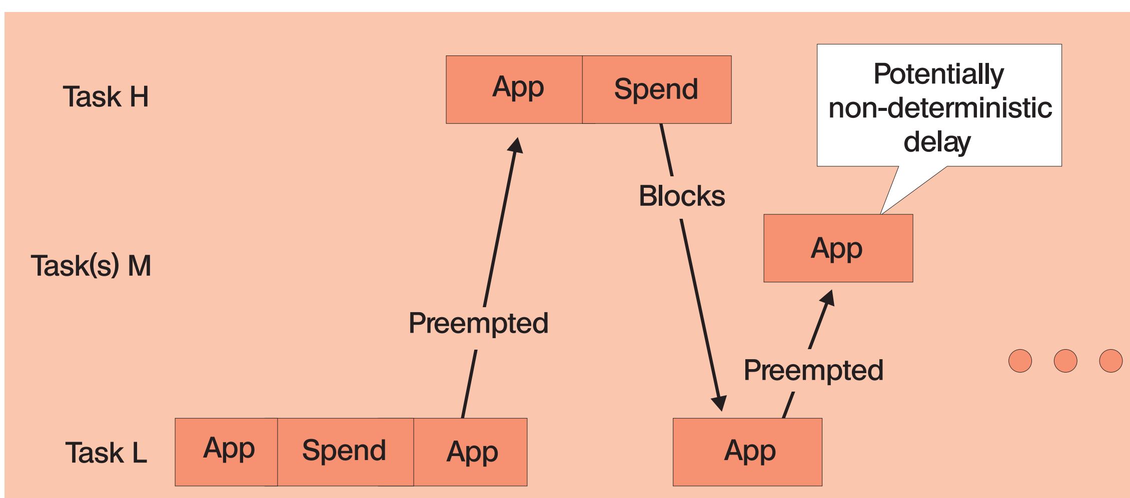

Figure 2. Priority Inversion.

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

AD VMETRO

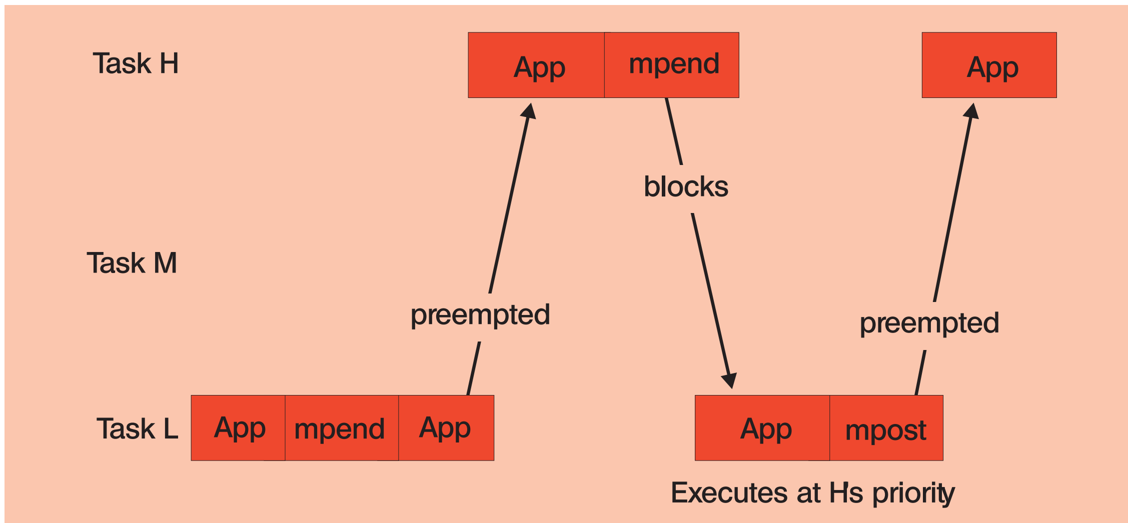

Figure 3. Priority Inheritance

services do not operate atomically. For example, if an RTOS's message queue implementation uses underlying semaphores, the RTOS itself can be subject to unbounded priority inversion if a high-priority task and a low-priority task attempt concurrent access to a queue.

AN ALTERNATIVE DESIGN WHICH MINIMIZES SCHEDULING LATENCY

A technique called "priority inheritance" provides the answer. Consider the same interaction between tasks H, L, and Mx described above, but with the introduction of a new mutual exclusion mechanism called a "priority inheritance mutex" in place of the traditional semaphore. (The term "mutex" has been popularized by the POSIX.1 standard.)

The new sequence of events is illustrated in Figure 3:

1. Low-priority task L locks priority inheritance mutex m
2. High-priority task H preempts L
3. H attempts to lock m
4. L resumes execution, but temporarily "inherits" H's priority
5. Medium-priority tasks Mx cannot preempt
6. L runs without interruption until it relinquishes m to H and returns to its original priority
7. H resumes execution

The use of priority inheritance protocols presents a neat solution to the priority inversion problem, resulting in more expedient handling of high-priority activities.

The use of priority inheritance eliminates the risk of unbounded priority inversion within a preemptable kernel. If mechanisms that implement priority inheritance are exported to the application as well, applications can be designed more easily with bounded priority inversion.

CONCLUSION

Including COTS software in your system has the potential to deliver great advantages in terms of improved time-to-market and increased robustness. However, those advantages will only be delivered if the COTS software has been developed and tested according to appropriate software engineering standards and designed as thoughtfully as you would design your own software, with the right feature set and architecture ■

Linda Thompson has been involved in the embedded software industry for over a decade, and has developed, sold, and marketed RTOS products. She currently works as Product Line Manager at Mentor Graphics' Embedded Software Division, where she has product marketing responsibility for the VRTX RTOS.

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)