The Truth About RTOS Overhead

March 20, 2010

Mark Slevinsky
Funkenstein
Software Consulting

Introduction

One of the main arguments against using an RTOS in an embedded project is that the overhead incurred is too great to be justified. Concerns over “wasted” RAM caused by using multiple stacks, added CPU utilization, and the “large” code footprint from the kernel cause a large number of developers to shun using a preemptive RTOS, instead favoring a non-preemptive, application-specific solution.

I believe that not only is the impact negligible in most cases, but that the benefits of writing an application with an RTOS can lead to savings around the board (code size, quality, reliability, and development time). While these other benefits provide the most compelling case for using an RTOS, they are far more challenging to demonstrate in a quantitative way, and are clearly documented in numerous industry-based case studies.

While there is some overhead associated with an RTOS, the typical arguments are largely unfounded when an RTOS is correctly implemented in a system. By measuring the true overhead of a preemptive RTOS in a typical application, we will demonstrate that the impact to code space, RAM, and CPU usage is minimal, and indeed acceptable for a wide range of CPU targets.

To illustrate just how little an RTOS impacts the size of an embedded software design we will look at a typical microcontroller project and analyze the various types of overhead associated with using a pre-emptive realtime kernel versus a similar non-preemptive event-based framework (both of which are available as part of FunkOS).

RTOS overhead can be broken into three distinct areas:

Code space: The amount of code space eaten up by the kernel (static)

Memory overhead: The RAM associated wtih running the kernel and application tasks.

Runtime overhead: The CPU cycles required for the kernel's functionality (primarily scheduling and task switching)

While there are other notable reasons to include or avoid the use of an RTOS in certain applications (determinism, responsiveness, and interrupt latency among others), these are not considered in this discussion – as they are difficult to consider for the scope of our “canned” application.

Application description:

For the purpose of this comparison, we first create an application using the standard preemptive FunkOS kernel with 2 system threads running: A foreground task and a background task. This gives three total priority levels in the system – the interrupt level (high), and two application priority threads (medium and low), which is quite a common paradigm for microcontroller firmware designs. The foreground task processes a variety of time-critical events at a fixed frequency, while the background task processes lower priority, aperiodic events. When there are no background task events to process, the processor enters its low-power mode until the next interrupt is acknowledged.

The contents of the tasks themselves are unimportant for this comparison, but we can assume they perform a variety of I/O using various user-input devices and a serial graphics display. As a result, a number of FunkOS device drivers are also implemented.

The application is compiled for an ATMega328p processor which contains 32kB of code space in flash, and 2kB of RAM, which is a lower-mid-range microcontroller in Atmel's 8-bit AVR line of microcontrollers. Using the WinAVR GCC compiler with -O2 level optimizations, an executable is produced with the following code/RAM utilization:

32600 Bytes Code Space
2014 Bytes RAM

An alternate version of this project is created using the FunkOS cooperative-mode kernel, which uses a single application thread and provides 2 levels of priority (interrupt and application). In this case, the event handler processes the different priority application events to completion from highest to lowest priority.

This approach leaves the application itself largely unchanged. Using the same optimization levels as the preemptive kernel, the code compiles as follows:

30904 Bytes Code Space
1648 Bytes RAM

Memory overhead:

At first glance, the difference in RAM utilization seems quite a lot higher for the preemptive mode version of the application, but the raw numbers don't tell the whole story.

The first issue is that the cooperative-mode total does not take into account the system stack – whereas these values are included in the totals for RTOS version of the project. As a result, some further analysis is required to determine how the stack sizes truly compare.

In cooperative mode, there is only one thread of execution – so considering that multiple event handlers are executed in turn, the stack requirements for cooperative mode is simply determined by those of the most stack-intensive event handler.

In contrast, the preemptive kernel requires a separate stack for each active thread, and as a result the stack usage of the system is the sum of the stacks for all threads.

Since the application and idle events are the same for both preemptive and cooperative mode, we know that their (independent) stack requirements will be the same in both cases.

For cooperative mode, we see that the idle task stack utilization is lower than that of the application task, and so the application task's determines the stack size requirement. Again, with the preemptive kernel the stack utilization is the sum of the stacks defined for both threads.

As a result, the difference in overhead between the two cases becomes the extra stack required for the idle task – which in our case is (a somewhat generous) 64 bytes.

The numbers still don't add up completely, but looking into the linker output we see that the rest of the difference comes from the extra data structures used to declare the tasks in preemptive mode.

With this taken into account, the true memory cost of a 2-thread system ends up being around 150 bytes of RAM – which is less than 8% of the total memory available on this particular microcontroller. Whether or not this is reasonable certainly depends on the application, but more importantly, it is not so unreasonable as to eliminate an RTOS-based solution from being considered.

Code Space Overhead:

The difference in code space overhead between the preemptive and cooperative mode solutions is less of an issue. Part of this reason is that both the preemptive and cooperative kernels are relatively small, and even an average target device (like the Atmega328 we've chosen) has plenty of room.

FunkOS can be configured so that only features necessary for the application are included in the RTOS – you only pay for the parts of the system that you use. In this way, we can measure the overhead on a feature-by-feature basis, which is shown in the table below:

Module	**Description**	**Size (bytes)**
driver	Device driver infrastructure	478
heap	Fixed-block dynamic memory	280
message	Simple Inter-process communications	386
mutex	Resource-protection locks	468
plumber	Complex inter-process communications	1054
semaphore	Thread synchronization object	446
task	FunkOS portable kernel	802
(port)	Port-specific functions	696
timer	Lightweight threads and periodic events	228
watchdog	Software watchdog interface	188
	All features	**5026**

The configuration tested in this comparison uses the **task/port** module with **timers**, **drivers**, and **semaphores**, for a total kernel size of ~2.6KB, with the rest of the code space occupied by the application.

The FunkOS cooperative-mode framework has a similar structure which is broken down by module as follows:

Module	Description	Size (bytes)
driver	Device driver infrastructure	272
ptask	FunkOS cooperative mode framework	854
timer	Lightweight threads and periodic events	202
	All features	**1328**

As can be seen from the compiler's output, the difference in code space between the two versions of the application is about 1.5kB – or about 5% of the available code space on the selected processor. While about 1.3kB of this is from the difference in the kernels, the rest of the difference comes the changes to the application necessary to facilitate the different frameworks.

Runtime Overhead:

On the cooperative kernel, the overhead associated with running the task is the time it takes the kernel to notice a pending event flag and launch the appropriate event handler, plus the timer interrupt execution time.

Similarly, on the preemptive kernel, the overhead is the time it takes to switch contexts to the application task, plus the timer interrupt execution time.

The timer interrupt overhead is similar for both cases, so the overhead then becomes the difference between the following:

Preemptive mode

-Posting the semaphore that wakes the high-priority thread
-Performing a context switch to the high-priority thread

Cooperative mode

-Setting the high-priority task's event flag
-Acknowledging the event from the event loop

Using the cycle-accurate AVR simulator, we find the end-to-end event sequence time to be 20.4us for the cooperative mode scheduler and 50.2us for the preemptive, giving a difference of 29.8us.

With a fixed high-priority event frequency of 33Hz, we achieve a runtime overhead of 983.4us per second, or 0.0983% of the total available CPU time. Now, obviously this value would expand at higher event frequencies and/or slower CPU frequencies, but for this typical application we find the difference in runtime overhead to be neglible for a preemptive system.

Analysis:

For the selected test application and platform, including a preemptive RTOS is entirely reasonable, as the costs are low relative to a non-preemptive kernel solution. But these costs scale relative to the speed, memory and code space of the target processor. Because of these variables, there is no “magic bullet” environment suitable for every application, but FunkOS attempts to provide frameworks suitable for a wide range of targets.

On the one hand, if these tests had been performed on a higher-end microcontroller such as the ATMega1284p (containing 128kB of code space and 16kB of RAM), the overhead would be in the noise. For this type of resource-rich microcontroller, there would be no reason to avoid using the FunkOS preemptive kernel.

Conversely, using a lower-end microcontroller like an ATMega88pa (which has only 8kB of code space and 1kB of RAM), the added overhead would likely be prohibitive for including a preemptive kernel. In this case, either the cooperative-mode kernel or the Pipsqueak kernel would be better choices.

As a rule of thumb, if one budgets 10% of a microcontroller's code space/RAM for a preemptive kernel's overhead, an AVR with 16kB of code space and 2kB of RAM would be a sufficient target as a minimum.

Conclusion:

After measuring and comparing the various types of overhead associated with using a preemptive RTOS in a system, we have demonstrated that the overhead added by a preemptive kernel is acceptable for all but the smallest of microcontrollers, and certainly not the major resource hog that the nay-sayers would have you believe.