

Multitasking—Essential to Any RTOS

[Electronic Design](#)

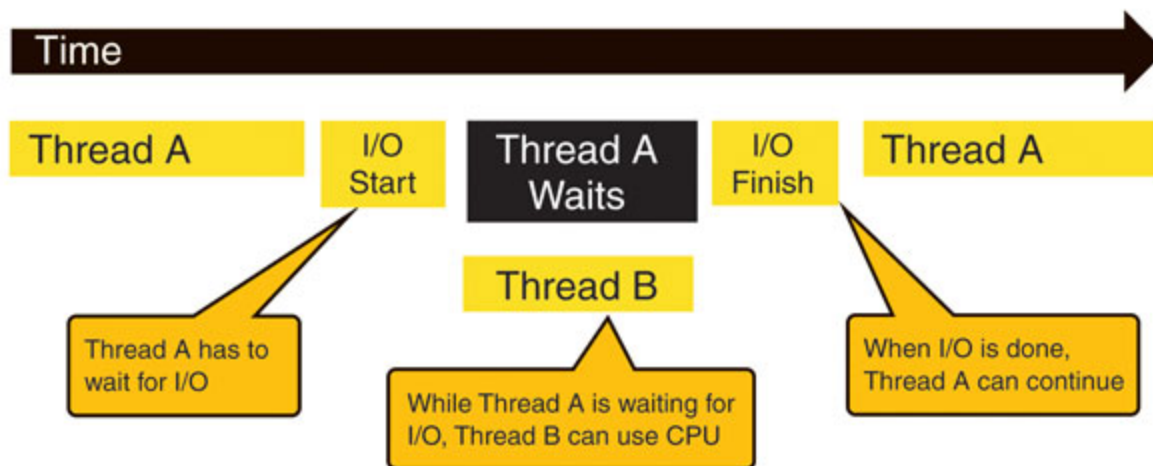
[John Carbone](#)

Mon, 2015-02-09 09:20



In 1997, Bill Lamie, author of the ThreadX RTOS, wrote an article entitled “*Multitasking Mysteries Revealed*.” In that article, Bill examined some of the benefits of multitasking, and the use of a real-time operating system (RTOS) for embedded real-time systems. Although written over 17 years ago, that article still accurately reflects the issues currently surrounding multitasking, and I think it would be beneficial to revisit this topic today.

The Motorola 68000 referenced in that article has given way to ARM processors, and SDS tools have been supplanted by IAR’s Embedded Workbench IDE. But the underlying principles of multitasking—its technology, benefits, and shortcomings—remain the same (*Fig. 1*).



Origins of Embedded Multitasking

Back in the “old days,” around the time of the UNIVAC and CDC mainframes, many significant periods of time were spent waiting to read data from a card reader or magnetic tape, or to output text to a printer or commands to a plotter. These were not real-time systems, but “batch” oriented services, designed for computing tasks such as mathematical computation, business operations, and database information storage and retrieval.


Within that same timeframe, embedded systems began to emerge, mostly for military applications. These required RTOSs or “executives.” Such systems could respond to real-time events and handle other tasks in the background while waiting for the next event. The early RTOSs employed foreground/background architectures, where a “Big Loop” type of sequential scheduler controlled the background and the foreground was a glorified ISR (*Fig. 2*).

```
main()
{
/*Look for high-priority serial input. */
pool_for_serial_input();

/*Some other program task "a." */
program_task_a();

/*Some other program task "b." */
program_task_b();

/*Some other program task "c." */
program_task_c();
}
```



Worst case
response time
to get back to
high-priority
serial polling

The Big Loop Scheduler became a problem. As memory expanded and applications grew, the loop expanded and responsiveness went on the decline. Multitasking offered a mixed mode of adjusted background scheduling, where real-time events could influence the scheduling of background tasks.

With multitasking, tasks would run until they were blocked or until a time limit expired. Then, while waiting for their next turn or until whatever was blocking them was gone, other tasks were allowed to run. This way, multiple tasks were active at any point in time, but at various stages of their code—only one actually executed instructions while the others waited their turn.

Multitasking proved to be a lot more efficient than the Big Loop. It became the backbone of all modern RTOS architectures and even multithreading hardware schedulers like those found in MIPS and Intel processors.

Multitasking More Critical Today than Ever

Technology and market trends in embedded computing continue to push demand for RTOS control. Specifically, larger memory, faster processors, smaller packaging, and the pervasiveness of networking, USB, and graphics have changed the landscape of embedded systems, and thus, RTOSs.

Related

[Express Logic ThreadX RTOS Is Extended To ARC 600 And 700 Processor Cores](#)

[Hardware-Based IP Speeds Up Processor Multitasking](#)

[“ThreadX With DAMs” For Cortex-M3/-M4 Delivers Remote-Update, Memory Protection](#)

Despite the many capabilities and services of an RTOS, some developers fear using them because of what they consider additional unnecessary overhead. It turns out that in most cases, the RTOS can actually reduce overhead through its more efficient scheduling algorithms compared to the Big Loop method, as is typically employed in non-RTOS applications.

Given the increased relevance and importance of an RTOS in today's embedded systems, it makes sense to

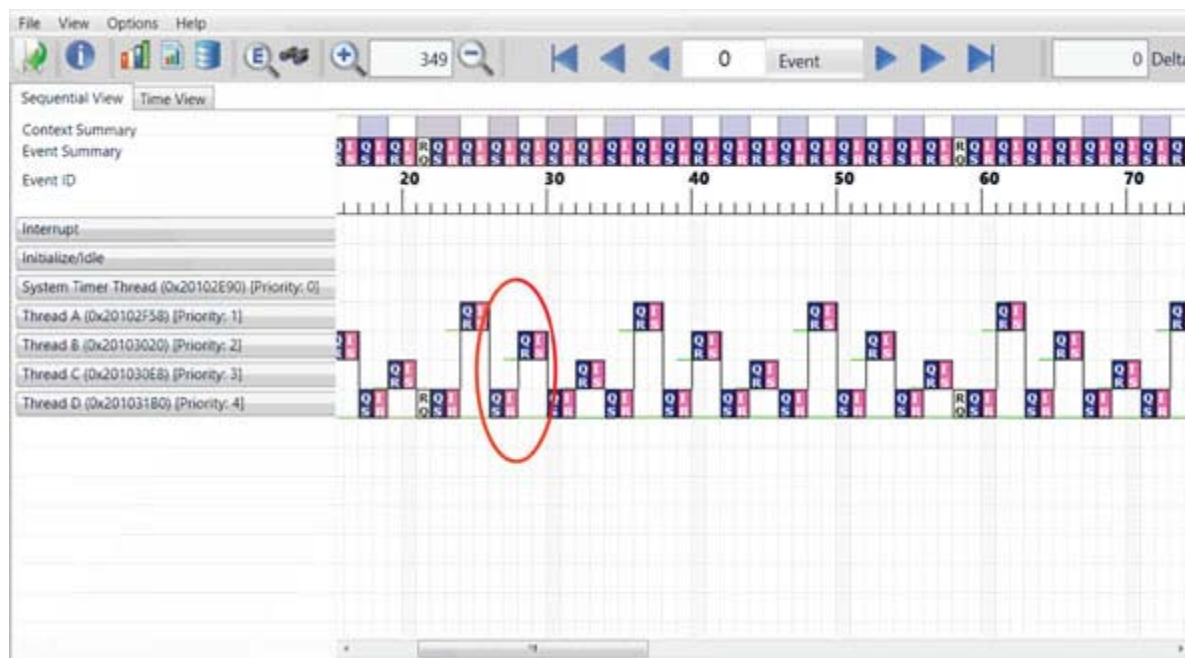
isit multithreading fundamentals to see if your next project might benefit from an RTOS.

So what's changed over the past 15 years? For one thing, the need for third-party middleware like TCP/IP stacks, USB, etc., has become mainstream, making it a necessity most of the time rather than an occasional need as was the case in 1997. This further accentuates the need for a multitasking foundation to make it easier and more efficient to add off-the-shelf middleware, rather than try to integrate it into a control loop structure.

The Benefits of Multitasking

Multitasking offers several benefits for embedded systems, especially as they grow beyond 32 or 64 kB of code:

- **Responsiveness:** Before multitasking was popular, most embedded applications allocated processing time with a simple control loop, usually from within the C main function. In large and/or complex applications, the response time to external events is directly affected by the entire control-loop processing. Having fast and deterministic response time allows application developers to concentrate on specific requirements of each application task without worrying about their effect on other system response times. Furthermore, modifying the program in the future becomes much easier because the developer needn't worry about affecting existing responsiveness with changes in unrelated areas (*Fig. 3*).



- **Ease of development:** In non-multitasking environments, each engineer should have intimate knowledge of the run-time behavior and requirements of the complete system. Multitasking, on the other hand, frees each engineer from the worries associated with processor allocation, allowing them to concentrate on their specific piece of the application software. Development teams can be assigned individual areas of responsibility, without having to worry about the others.

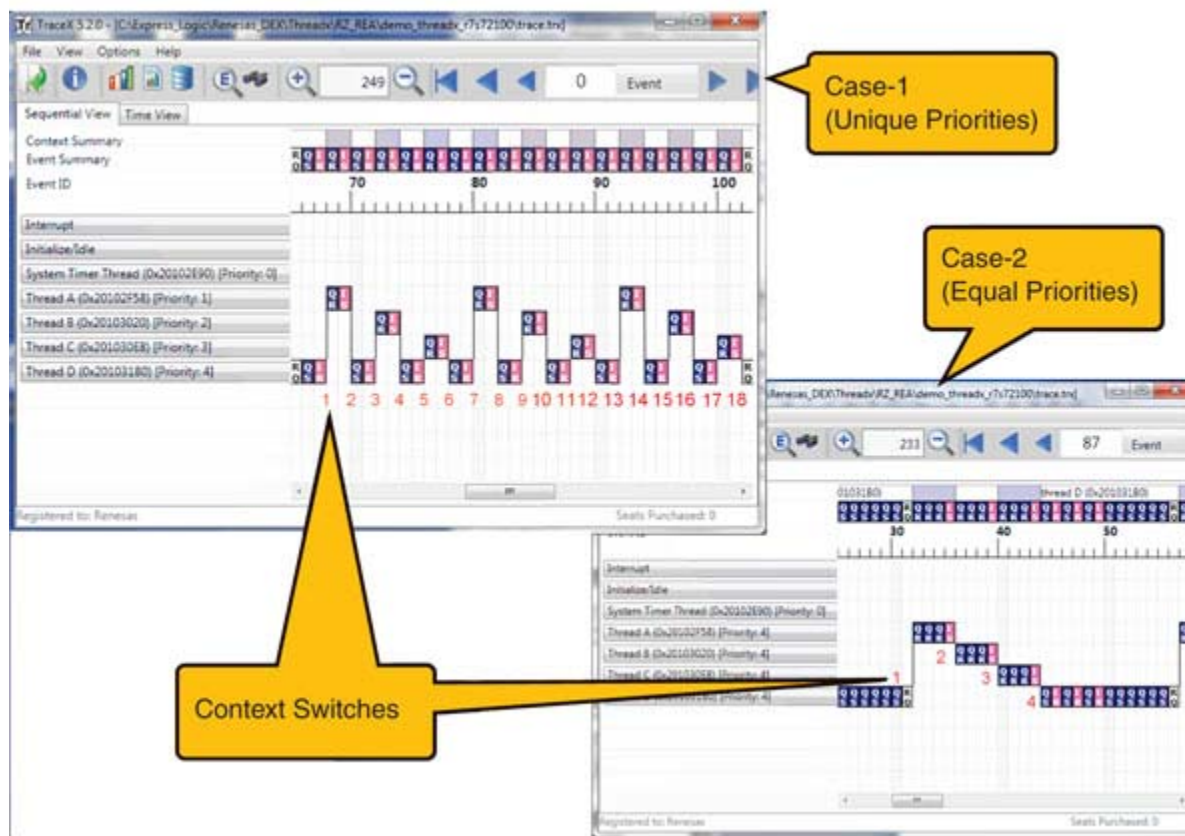
- **Portability and maintenance:** Most multitasking operating environments provide a layer of abstraction between the underlying processor and the application. Processor-specific activities like context switching and interrupt processing are inherently handled by the multitasking environment. This makes applications much easier to port to new processors, and much easier to modify and maintain.

Common Pitfalls

While multitasking offers many attributes, it also invites certain dangers that the developer must be careful to

id. The biggest problem? Misuse of multitasking translates into excessive memory usage, unwanted run-time behavior, and excessive processor overhead.

- **Memory pitfalls:** Multitasking run-time environments almost always require more memory than the simple control-loop run-time environments.
- **Priority pitfalls:** The main caution here is to avoid different priorities as much as possible, i.e., applications should be designed so that threads are only assigned a higher priority when they absolutely *must* preempt threads of lower priority.
- **Excessive overhead:** One often-overlooked way to shrink overhead in multitasking environments is to reduce the number of context switches. This sounds like a great idea, but how is it done? The first technique is to simply limit the number of tasks. The fewer tasks in the system, the fewer context switches. Although reducing the number of tasks is a good start, it's the task priorities that really need close attention in this scenario (Fig. 4).



Buy vs. Build

Assuming that multitasking is beneficial to your application, the next question is whether to buy a kernel or build it yourself. Building the environment yourself makes it possible to tailor the environment to your specific needs. However, most custom environments require significant development time; are more expensive; and are less functional and portable than their commercial alternatives.

In general, it's probably better to buy the right commercial multitasking product (it will cost around three to four weeks of your salary). Also, using a commercial product allows your team to concentrate on the actual application instead of the run-time environment, which helps get your product to market faster. Because commercial multitasking products have to support many different processor families to stay in business, your

ware investment is protected.

Summary

Hopefully, this article will help you determine when to use multitasking, and if you do decide to use multitasking, how to achieve the most benefit from using it. If you're not yet using multitasking (a significant number of embedded applications still haven't reached that point), save this article someplace safe because you probably will need it soon!

John A. Carbone, vice president of marketing for [Express Logic](#), has 35 years' experience in real-time computer systems and software, ranging from embedded system developer and FAE to vice president of sales and marketing. He holds a BS in mathematics from Boston College.

Source URL: <http://electronicdesign.com/embedded/multitasking-essential-any-rtos>