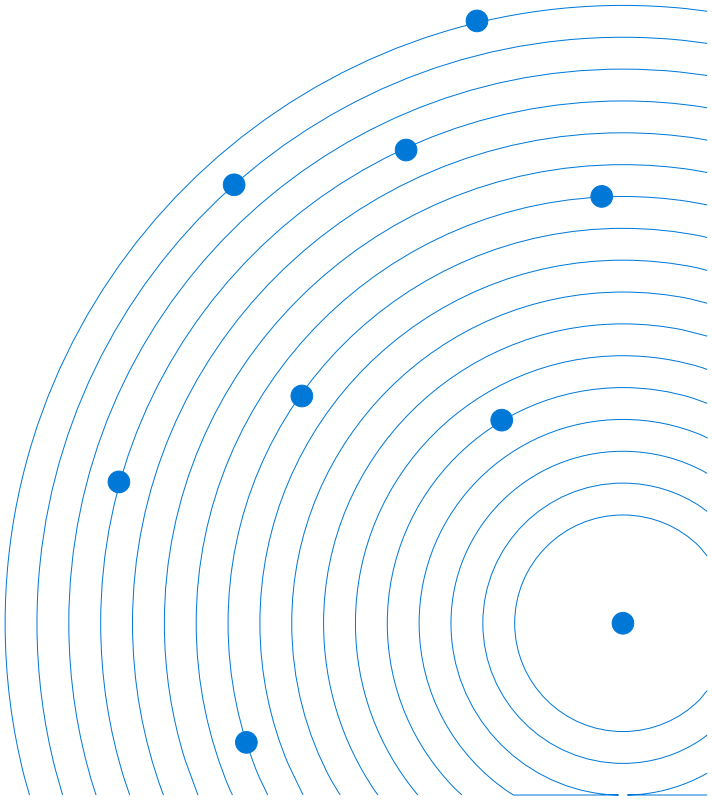
Real-Time Operating System

What it is and why you might want to use one

January 2020



Microsoft

Contents

Introduction 3

What exactly is a Real-Time Operating System? 4

When do you need a Real-Time Operating System? 5

What are the benefits of an RTOS? 6

- Benefit #1: RTOS Provides Fast, Guaranteed Real-Time Performance 6
- Benefit #2: An RTOS Can Reduce Overhead 8
- Benefit #3: An RTOS Eases Development 8
- Benefit #4: An RTOS Makes it Easier to Add New Features 9
- Benefit #5: Easier Application Portability 9
- Benefit #6: Safety Certification 9

What are the costs of an RTOS? 10

- Cost #1: A Good RTOS Requires Purchase 10
- Cost #2: An RTOS Does Require Memory 10
- Cost #3: An RTOS Does Require Cycles 10
- Cost #4: Learning to Use an RTOS 10
- Cost #5: An RTOS Can Be Overkill 11
- Cost #6: Buy vs. Build 11

Summary 11

Introduction

Industry estimates predict there will be over 20 Billion Internet of Things (IoT) devices shipping by the year 2020 (source: Gartner.com). Most IoT devices will involve some type of network connectivity (e.g., Wi-Fi, Bluetooth LE, Zigbee, 6LoWPAN, ethernet), and many also will include a Graphical User Interface (GUI). These devices usually also perform some functions other than network communication and user interaction, such as data acquisition (sensors), appliance control (home automation), and DSP (security systems). Because of this processor load, all but the simplest IoT devices will require 32-bit microprocessors or microcontrollers in order to provide the necessary address space and processing power. There already is strong migration from 8-bit and 16-bit to 32-bit microprocessors, due to requirements for enhanced device functionality as well as the attractive cost/performance attributes of new 32-bit microprocessors. The predicted IOT explosion promises to sharply accelerate this migration!

The migration to 32-bit processors is clear, but what about the software side? The increased connectivity requirements alone necessitate the execution of communication protocol stack(s) on the embedded microprocessor, which in turn necessitates the use of a Real-Time Operating System (RTOS). GUI design and runtime software from 3rd parties typically rely on RTOS services as well. Other functions add complexity to the application, and warrant multithreading to handle both processing and real-time response. With the rapid growth of the IoT and the number of new devices being developed to take advantage of it, it's becoming more and more likely that you will consider using an RTOS in the near future. The remainder of this article will explore the overall RTOS value proposition, both pros and cons, so you can consider whether an RTOS would be beneficial for your next product development project.

What exactly is a Real-Time Operating System?

A Real-Time Operating System is system software that provides services and manages processor resources for applications. These resources include processor cycles, memory, peripherals, and interrupts. The main purpose of a Real-Time Operating System is to allocate processing time among various duties the embedded software must perform. This typically involves a division of the software into pieces, commonly called "tasks" or "threads," and creating a run-time environment that provides each thread with its own virtual microprocessor ("Multithreading"). Basically, a virtual microprocessor consists of a virtual set of microprocessor resources, e.g., register set, program counter, stack memory area, and a stack pointer. Only while executing does a thread use the physical microprocessor resources, but each thread retains its own copy of the contents of these resources as if they were its own private resources (the thread"s "context").

The Real-Time Operating System controls thread execution, and the accompanying management of each thread"s context. Each thread is given a priority by the designer, to control which thread should run if more than one is ready to run (ie: not blocked). When a higher-priority thread (compared to the running thread) needs to execute, the RTOS saves the currently running thread"s context to memory and restores the context of the new thread. The process of swapping the context of threads is commonly called context switching.

Figure 1 shows example context switching code for the Cortex-M architecture. Typical context switching on a Cortex-M requires less than 120 cycles, i.e., less than 1µs on a 120MHz Cortex-M.

```
_save_thread_context:
    MRS R12,PSP                 ; Pickup thread's stack pointer
    STMDB R12!,{R4-R11}         ; Save thread context
    LDR R0,=_current_thread     ; Pickup current thread pointer address
    LDR R1,[R0]                 ; Pickup current thread pointer
    STR R12,[R1]                ; Save thread's stack pointer
    MOV R12,#0                  ; Build NULL value
    STR R12,[R0]                ; Clear current thread pointer
    B _scheduler                ; Look for next thread to execute
...
_scheduler:
    LDR R0,=next_thread         ; Pickup next thread pointer address
    LDR R1,[R0]                 ; Pickup next thread to schedule
    CBZ R1,_scheduler           ; If no thread, continue checking
    B _restore_thread_context
...
_restore_thread_context:
    LDR R0,=_current_thread     ; Pickup current thread pointer address
    STR R1,[R0]                 ; Setup current thread pointer
    LDR R12,[R1]                ; Pickup thread's stack pointer
    LDMIA R12!,{R4-R11}         ; Recover thread context
    MSR PSP,R12                 ; Setup thread stack pointer
    MOV LR,#0xFFFFFFFD          ; Setup return to thread on PSP
```

Figure 1: An RTOS context switch is often more efficient than a polling loop. Compared to Express Logic's ThreadX, a polling loop would have to require fewer than 120 cycles or 60 instructions – a real challenge when all functions must be checked and executed.

This transfer of control to another thread is immediate and invisible to the embedded software. In fact, this invisibility might be the most fundamental benefit of an RTOS. Instead of embedding processor allocation logic inside the application software, it's done by the RTOS. This arrangement isolates the processor allocation logic and makes it much easier to predict and adjust the run-time behavior of the embedded device.

It's important to note that an RTOS must offer preemption. Preemption is the action of switching to a higher-priority thread instantly and transparently, without having to wait for completion of the lower-priority thread. In addition to processor allocation, a good commercial RTOS provides additional communication, synchronization, and memory allocation services. *Figure 2* shows an example of a typical, commercial RTOS.

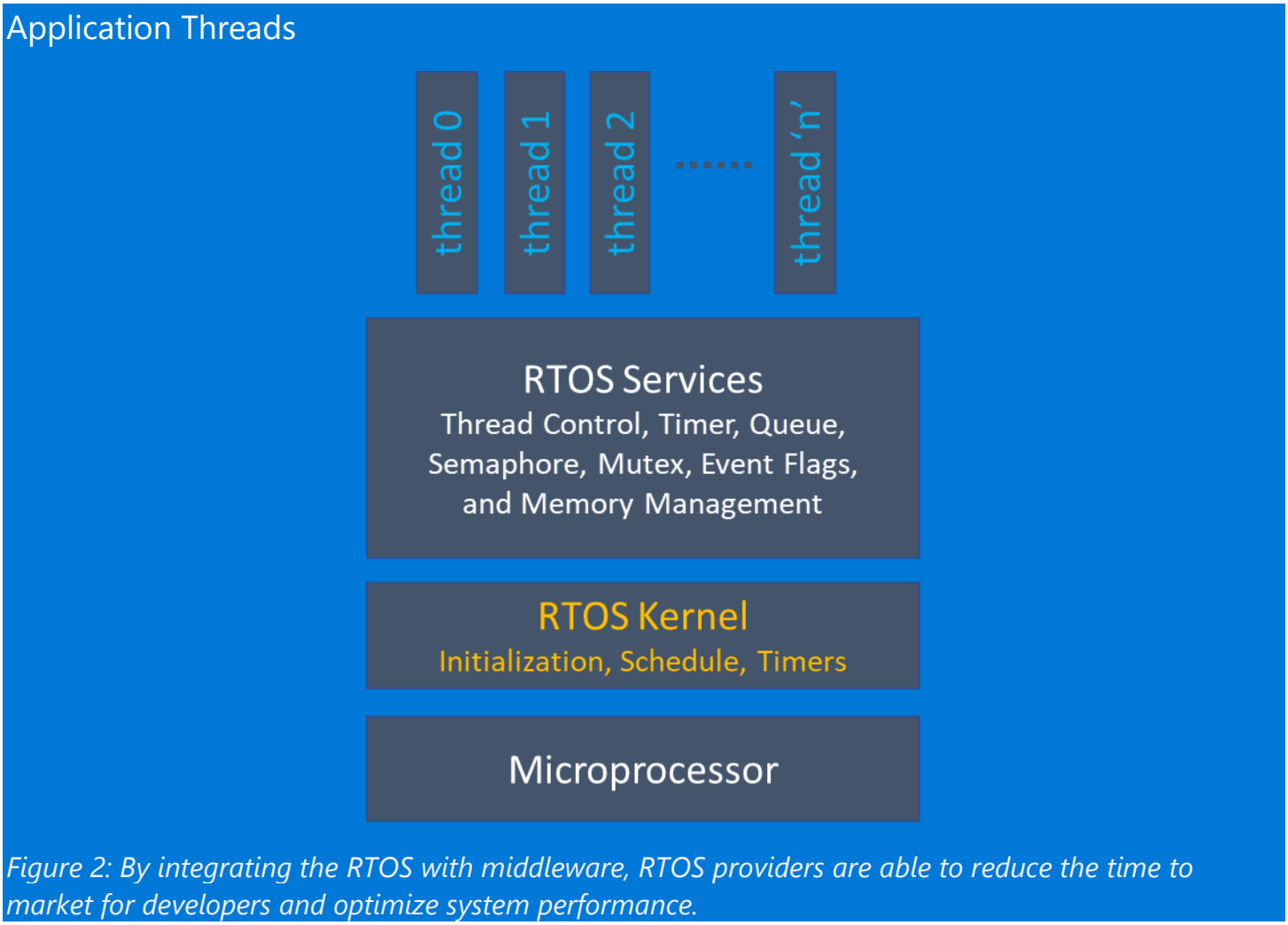


Figure 2: By integrating the RTOS with middleware, RTOS providers are able to reduce the time to market for developers and optimize system performance.

When do you need a Real-Time Operating System?

Need is a strong word – in reality, we don't really need many of the things we take for granted – they just make life easier and more enjoyable. With enough development effort, a team can write their own software to handle all required application functions (in effect, a DIY RTOS). The issue is how much time will that take? Will it perform as well as a commercial RTOS? Is support going to be a burden? What about documentation? These are some of the key questions that must be answered to properly compare a DIY RTOS approach to using a commercial RTOS. Even if you don't "need" an RTOS, it may still be beneficial to your product development and your product's eventual success.

So, in this paper, we will try to highlight the benefits of using an RTOS, and the related costs. it's up to each development team to assess the weight of each side, so that the best choice for your project can be made. Even if you don't need an RTOS, the benefits might outweigh the cost and make it a smart choice.

What are the benefits of an RTOS?

There are many benefits of using an RTOS, which is why more and more embedded devices are using them. In fact, according to the most recent UBM Embedded Developer survey, released in April, 2017, over 59% of current projects require real-time capabilities, over a third use a GUI, and as a result, over 67% report using an RTOS or scheduler of some kind. Among the remaining 33% who didn't use an RTOS, the biggest reason for not using one (86%) was that the application didn't "need" it. Of those who selected a commercial RTOS, 45% cited the #1 reason as "Real-Time Capability." To decide whether YOUR application might benefit from the use of an RTOS, here are some potential benefits to consider:

Benefit #1: RTOS Provides Fast, Guaranteed Real-Time Performance

An RTOS invisibly handles allocation of the processor's resources to the threads that perform the various duties an embedded device must perform. With the proper assignment of thread priorities, the application software does not have to concern itself with how much processing time each individual thread takes. Even better, changes in the processing responsibilities of a given thread do not impact the processor allocation (responsiveness) of higher-priority threads.

Before multithreading was popular, most embedded applications allocated processing time with a simple control loop usually from within the C main() function. Figure 3 shows a simple polling loop that examines a real-time sensor and then performs three additional functions. The developer has complete access and understanding of this closed, simple system such that the real-time interrogation of the external sensor can be guaranteed. In large and/or complex applications, the response time to external events is directly affected by the entire control loop processing.

```
main()
{
       /* Enter polling loop. */
       while(1)
   {
       /* Check external sensor. */
       check_external_sensor();

       /* Perform first firmware function. */
       first_function();

       /* Perform second firmware function. */
       second_function();

       /* Perform third firmware function. */
       third_function();
   }
}
```

Figure 3: This simple polling loop examines a real-time sensor and then performs three additional functions.

Having fast and deterministic response time allows application developers to concentrate on specific requirements of each application function without worrying about what affect they might have on other system response times. Furthermore, modification of the program in the future is much easier because the developer does not have to

worry about affecting existing responsiveness with changes in unrelated areas. Many legacy 8/16-bit devices employ a polling loop software architecture to distribute processing time among its various functions. Although easy to understand and appropriate in very simple devices, this approach suffers greatly when used in more complex devices.

The fundamental problem of a polling loop is that the responsiveness of any thread in the loop is determined by processing elsewhere in the polling loop. The worst-case responsiveness is effectively the worst-case processing through each function in the loop. If processing in the polling loop changes, so does the responsiveness of each thread. As greater complexity is added to the polling loop, the more difficult it becomes to predict and meet real-time requirements. Take for example an existing device that must sample some external sensor in real-time. If this example device then needs to be connected to the cloud, the polling loop would need to be expanded to also invoke a software protocol stack for the cloud communication.

Figure 4 shows the polling loop with an additional call, "perform_cloud_communication." Adding this extra processing to the polling loop might reduce the responsiveness to a level where the external sensor is no longer sampled properly. In addition, communication protocols often range from 50KB to 100KB in size, which indicates a level of complexity that might make it difficult to calculate its worst-case processing requirement.

```
main()
{
       /* Enter polling loop. */
       while(1)
   {
       /* Check external sensor. */
       check_external_sensor();

       /* Perform first firmware function. */
       first_function();

       /* Perform second firmware function. */
       second_function();

       /* Perform third firmware function. */
       third_function();

       /* Perform Cloud Communication */
       perform_cloud_communication();
   }
}
```

Figure 4: Here the polling loop makes an additional call to the cloud. Just adding this one call can reduce responsiveness enough that the external sensor would no longer be correctly sampled.

In sharp contrast, the response time using an RTOS is constant. In the same example, the protocol stack processing could be placed in a lower-priority thread, while the real-time sensor sampling could be placed in a higher-priority thread. In that case, the RTOS would ensure that the real-time sensor sampling thread always takes precedence over the communication thread. This results in completely deterministic and faster response times for the external sensor sampling thread. And of course, this is all done by the RTOS, invisible to the application software.

Benefit #2: An RTOS Can Reduce Overhead

This benefit often is misunderstood. Many developers believe that an RTOS adds overhead. After all, it's additional code beyond that of the application. Accordingly, they may feel that they "cannot afford the overhead of using an RTOS." This additional code must take some time to execute, thereby delaying real-time response. However, when compared to the alternative - using a polling loop to manage thread execution and responsiveness - an RTOS often results in lower overhead and better responsiveness. Here is how that alternate solution might look, compared to the use of an RTOS.

The alternate strategy to achieve better polling loop responsiveness is to simply add more polling. By placing more polling throughout the code, some of the responsiveness problems can be solved. This can improve responsiveness, but this approach increases overhead, which drains power and decreases throughput. Worse, it requires the developer to tackle more and more issues of context switching and preemption, both of which are already solved by an RTOS. Although an RTOS does introduce some overhead (its function calls and context switching certainly consist of some code), the amount of overhead is small and constant. This overhead is optimized over years of RTOS use and becomes about as efficient as is possible. On the other hand, once a simple polling loop becomes more complex, it's likely the overhead in the polling loop will exceed that of an optimized RTOS. For example, RTOS context switching on a Cortex-M typically takes less than 120 cycles (this can vary from architecture to architecture and from RTOS to RTOS). In order for a polling loop (or other alternative to an RTOS) to do better, it would have to be able to predict and guarantee that the worst-case delay in activating a thread with something to do would be less than 120 cycles (which is about 50-60 lines of C code). That means that the time taken to check each thread in the loop, and to execute the code to get to any thread that has something to do, would have to be less than 60 instructions total. Even if no other thread had anything to do, just checking each thread would consume cycles, until the thread that requires servicing is checked. For loops with 5-10 threads, this might work satisfactorily (again, assuming that no intervening thread has work to do). Once the loop gets more lengthy, those 60 instructions become inadequate to get all the way around. And, in a worst-case analysis, ALL the intervening threads must be given processing time, making real-time response virtually impossible even for a minimal 2-thread loop. Rather than dismissing the use of an RTOS because it contains overhead, be sure to compare the application code that would have to be used instead. You may be surprised to find out that the RTOS code is actually faster.

Benefit #3: An RTOS Eases Development

In non-multithreading environments, each engineer is required to have intimate knowledge of the run-time behavior and requirements of the complete system. Each firmware developer must have this knowledge because the processor allocation logic is dispersed throughout the application code. Small device firmware projects (typically less than 32KB of total memory) can be reasonably well handled by one or two firmware developers. Given the relatively small code size, the developers have a fighting chance to understand the processing requirements of all the firmware. However, as functionality of the device increases, the development team must be expanded and the knowledge of all the firmware processing requirements naturally becomes less well known. Communication among the code modules developed by each team member must then be designed and implemented, to allow for inter-thread synchronization and information exchange. Multithreading, on the other hand, frees developers from the worries associated with processor allocation and allows 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 other areas. Moreover, in an RTOS, they have inter-thread communication services available (eg: RTOS messaging, semaphores, etc.) that are efficient, consistent, and well-defined.

Middleware and the Real-Time Operating System

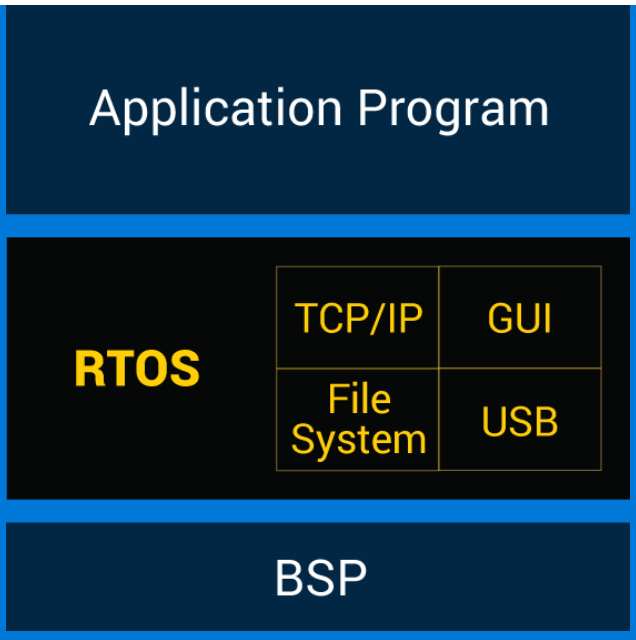


Figure 5: A commercial RTOS with Integrated Middleware

Benefit #4: An RTOS Makes it Easier to Add New Features

An RTOS invisibly handles the processor allocation logic, such that real-time performance of a high-priority thread can easily be guaranteed — regardless of whether the firmware is 32KB or 1MB in size, and regardless of the number of threads in the application. This alone makes it easier to maintain the application and easier to add new features to a device. In addition, most commercial RTOS offerings also have an extensive set of middleware that is pre-integrated and ready to be deployed. Having the ability to easily add networking, file systems, USB, and graphical user interfaces makes adding new features to a device that much easier. *Figure 5* shows the association of middleware and the RTOS.

Benefit #5: Easier Application Portability

Most multithreading operating environments provide a layer of abstraction between the underlying processor and the application. Processor specific activities like context switching and interrupt processing are handled by the multithreading environment. This makes applications much easier to port to new processors and maintain, as long as the underlying RTOS also supports that processor.

Applications that use an RTOS access its service functions through an Application Programming Interface (API). The API makes the RTOS platform independent - meaning that it's the same API regardless of what processor it's running on. That makes switching processors easier, since none of the application's service calls have to be changed. The application will run anywhere the RTOS can run. With most popular commercial and open source RTOSes, that means virtually any 32-bit processor architecture. This gives developers the benefit of application portability with minimal changes to their code.

Benefit #6: Safety Certification

Many safety-critical systems require certification by responsible government or other regulatory authorities. This generally requires that the system developer provide "artifacts" for all the software in the system. Many commercial RTOSes are "pre-certified" by these regulatory authorities, and that offers huge benefits to developers. By using a pre-certified RTOS, no artifacts need to be provided for the RTOS, speeding the process of certification

of the system. Without using a pre-certified RTOS, developers must either qualify their own scheduling and service code or create artifacts for the non-pre-certified RTOS they are using, often at a cost and additional time.

What are the costs of an RTOS?

There are many very important benefits to using an RTOS, many of which we have just described, but there also are costs associated with RTOS use. You should weigh both for your particular situation, and then decide whether using an RTOS would be best for you. Here are several costs that should be considered:

Cost #1: A Good RTOS Requires Purchase

The old adage of "you get what you pay for" applies to RTOS products just like everything else. The starting point for most commercial RTOS licenses is on the order of $10K for a royalty-free license with full source code and full support. This represents a tremendous value – only costing 1-2 months of a typical embedded firmware developer's salary. And it's a one-time investment since the license can be used forever, while that developer costs $5K - $10K each and every month. It's important to realize that "free" or open source RTOS products are not totally free, but simply free to use. Without a revenue stream, a free RTOS does not generally benefit from R&D, product evolution, and most importantly the ability to fully support customers. An RTOS without dedicated support is a like using "1234" as your password for everything – it isn't very smart, and it will eventually catch up to you! While a commercial RTOS does have a cost associated with its licensing for use, it might help you avoid other costs that you might encounter by using a "free" RTOS.

Cost #2: An RTOS Does Require Memory

An RTOS requires both instruction memory (usually flash memory) and RAM memory for its operation. However, a good commercial RTOS requires very little of either, typically on the order of 2KB of instruction area memory and 1KB of RAM. Most commercial RTOSes only link in the code for functions that are actually used by the application, so that the actual cost is not the size of the entire RTOS library. Of course, application needs in these areas also must be considered. Application code to perform the functions that otherwise would be handled by an RTOS also requires flash or ROM, as well as RAM. So, the incremental cost of an RTOS in terms of memory might not be as significant as it first appears.

Cost #3: An RTOS Does Require Cycles

An RTOS requires processing cycles for context switches, API calls, etc. This overhead typically concerns developers. Before concluding that this overhead is too expensive, consider two points. First, the processing cycles required are directly related to what the application uses. For example, if an existing polling loop were executed (as-is) from a single thread in an RTOS, there would be no additional overhead. The polling loop would execute just as before. The RTOS processing overhead only happens when RTOS services are utilized, such as context switching, message passing, and the like. Without an RTOS, the application is responsible for all required processing, including the cycles used in polling, function calling, messaging, and interrupt servicing. Aside from the simplest applications, it's likely that the RTOS cycles will turn out to be fewer than what the application would end up consuming if it had to do all the work. Second, a commercial RTOS is professionally optimized to be as efficient as possible, likely more efficient than fresh application code to perform the same functionality without an RTOS. While an RTOS has an overhead cost, a non-RTOS application might have even greater overhead cost.

Cost #4: Learning to Use an RTOS

Using an RTOS requires some learning. However, this too is proportional to what the application uses. In the previous example of just placing a legacy polling loop inside a single thread, there would be virtually no learning

necessary. Having said that, most good commercial RTOS companies provide on-site training as well as a substantial amount of professional documentation (textbook, user guide, etc.). Of course, a good commercial RTOS is designed to be simple to understand and use. Going back to the pitfalls associated with using a polling loop architecture, the amount of time studying and re-studying the performance of the entire firmware will quickly dwarf any time spent learning how to use an RTOS, especially for a non-author of the original code.

Cost #5: An RTOS Can Be Overkill

While it's hard to precisely state the criteria for "needing" an RTOS, as a rough guideline, when the total memory (ROM and RAM combined) of a device is less than 16KB, there is a decent chance that using an RTOS is overkill. Such devices typically have a single dedicated purpose and most often use an 8-bit or 16-bit microprocessor. Once device firmware exceeds 32KB of total memory and/or utilizes a communication protocol (like all IoT devices will) or GUI, it will almost certainly "need" an RTOS, or at least greatly benefit from the use of one. Even a small 32KB, dedicated purpose, non-IoT device could benefit by using an RTOS by simply isolating foreground and background processing into two separate threads. This configuration would make device firmware much simpler and much easier to enhance in the future.

Cost #6: Buy vs. Build

Assuming that multithreading is beneficial to your application, the next question is whether to buy an RTOS or build it yourself. Building a "scheduler" yourself allows you to tailor the environment to your specific needs. It also creates in-house experts for your code who can provide support going forward. However, a DIY approach involves "re-inventing the wheel" in the non-application area, which is not directly beneficial to the real value of the product. Most custom environments require significant development time, are therefore more expensive and less portable than their commercial alternatives. Using a commercial RTOS product allows your team to concentrate on the value of the actual application instead of the run-time environment, which helps make your product better and get it to market faster. Because commercial RTOSes support many different processor families, your software investment is protected. In general, it might be better to buy a good commercial RTOS and let your developers focus on your domain expertise.

Summary

With the continued migration to 32-bit microprocessors and billions of new IoT devices coming to market in the next several years, there is a strong case for using a commercial RTOS. Coupled with the relatively small cost of using a commercial RTOS, it is practically a foregone conclusion that an RTOS is in your near future – if you aren't using one already!

Microsoft