print | close

11 Myths About Using an RTOS in IoT Devices

Electronic Design

William Lamie

Thu, 2016-03-17 09:37

According to the 2015 UBM Embedded Developer survey, more than 60% of current projects include real-time capabilities, more than a third include a GUI, and more than 70% report using an RTOS or scheduler of some kind. Among the remaining 30% who didn't use an RTOS, the main reason for not using one (79%) was that the application didn't need it. But myths prevail about the reasons for using (or not using) an RTOS. Here are our top 11 myths.

1. Most IoT devices don't need an RTOS.

The embedded industry is already seeing strong migration away from 8- and 16-bit microprocessors due to requirements for enhanced device functionality as well as the attractive cost/performance attributes of new 32-bit microprocessors. IoT devices specifically require network connectivity and many will also include a graphical user interface (GUI). As a result, they generally require 32-bit microprocessors to provide the necessary address space and processing power.

Related

Q&A: Express Logic's X-Ware Platform Exhibits IoT Prowess

Multitasking—Essential to Any RTOS

Dev Kits: Getting Synergy

Similar evolutions are occurring on the software side. Increased connectivity requirements alone necessitate the execution of communication protocol stacks on the 32-bit embedded microprocessor, which in turn necessitates the use of an RTOS. GUI design and runtime software from third parties typically rely on RTOS services as well.

2. A polling-loop architecture works just as well as an RTOS.

Many legacy 8- and 16-bit devices employ a polling-loop software architecture to distribute processing time among their various threads.

Although easy to understand and appropriate in very simple devices, this approach suffers when used in more complex devices. The problem is that responsiveness of any thread in the loop is determined by the processing in the rest of the polling loop, making the worst-case responsiveness the worst-case processing through the polling loop. If processing in the polling loop changes dynamically, 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, which can impact the reliability of IoT devices.

In contrast, the response time using an RTOS is constant. Moreover, the RTOS invisibly handles allocation of

processor to thread priorities, so the application software needn't address how much processing time is en by each thread. Even better, changes in the processing responsibilities of a given thread don't impact the responsiveness of higher-priority threads.

3. An RTOS adds overhead that my IoT device can't afford.

Although an RTOS does introduce some overhead in API calls and context switching, the amount of overhead is small and constant and is likely less than a complex polling loop *(see figure)*. For example, RTOS context switching on a Cortex-M typically takes less than 120 cycles (this can vary from architecture to architecture and RTOS to RTOS).

In order for a polling loop (or other RTOS alternative) to do better, it would have to be able to predict and guarantee that the worst-case delay in activating a thread 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 execute the code for any active thread would have to be less than 60 instructions.

Even if no other thread had anything to do, just checking each thread would consume cycles, until the thread that requires servicing is checked. This might work for loops with 5-10 threads, but once the loop lengthens, those 60 instructions are inadequate. In a worst-case analysis, all of the intervening threads must be given processing time, making real-time response virtually impossible even for a minimal two-thread loop.

```
_save_thread_contex:
      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_contex:
      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
```

4. An RTOS makes development more complex.

Small device firmware projects (typically less than 32 kB of total memory) can be reasonably managed by one or two firmware developers who both must understand the run-time behavior and requirement of the device in total. That's because the processor allocation logic is dispersed throughout the application code. However, steadily increasing functionality of the device, such as adding cloud communication protocols, expands the development team and not everyone understands the firmware processing requirements. Communication

ong the code modules developed by each team member must then be designed and implemented to allow for er-thread synchronization and information exchange.

In contrast, an RTOS eases development when adding device functionality. With an RTOS, firmware developers can concentrate on their specific piece of the firmware and not have to worry about the processing requirements of the other firmware in the device. What's more, they have inter-thread communication services (messaging, semaphores, etc.) that are efficient, consistent, and well-defined.

5. An RTOS makes adding new features to my device more difficult.

An RTOS invisibly handles the processor allocation logic so that real-time performance of a high-priority thread can easily be guaranteed—whether the firmware is 32 kB or 1 MB 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. Also, most commercial RTOS offerings have an extensive set of pre-integrated middleware that makes it easy to add networking, file systems, USB, and graphical user interfaces.

6. An RTOS makes application portability more difficult.

Applications that use an RTOS access its service functions through an API, which makes the RTOS platform-independent.

That makes switching processors easier, since none of the application's service references have to be changed. The application will run anywhere the RTOS can run. With most popular commercial and open-source RTOSs, that means virtually any 32-bit processor architecture. This gives developers the benefit of application portability with minimal changes to their code.

7. An RTOS requires too much memory.

An RTOS does require 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 2 kB of instruction area memory and 1 kB of RAM. Of course, application needs in these areas also must be met, making it even more beneficial if the RTOS uses small amounts of memory for its operation.

8. An RTOS requires too many processing cycles.

While an RTOS requires processing cycles for performing context switches, executing API calls, etc., the processing cycles are directly related to what's used by the application. For example, if an existing polling loop is executed (as-is) from a single thread in an RTOS, there's no additional overhead. The polling loop would execute just as before.

RTOS processing overhead only happens when RTOS services are used. Also, without an RTOS, the application is responsible for all required processing, including the cycles used in polling, function calling, and interrupt servicing. Aside from the simplest applications, it's likely that the RTOS cycles will turn out to be less than the application would consume if it had to do all of the work.

9. I don't have time to learn an RTOS.

Learning to use an RTOS is proportional to what's used in the application. For example, placing a legacy polling loop inside a single thread would require virtually no additional learning. But with 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. Having said that, most good commercial RTOS companies provide on-site training as well as a

stantial amount of documentation. And, of course, a good commercial RTOS is also simple to understand l use!

10. I can't afford a commercial RTOS.

The old adage "you get what you pay for" applies to RTOS products just like everything else. Free or open-source RTOS products simply don't have the vested self-interest in being small, efficient, and easy to use. Not having a revenue stream also implies a lack of R&D, product evolution, and most importantly, the ability to fully support customers. An RTOS without dedicated support is like using "1234" as your password for everything—it might work, but it isn't very smart and it will eventually catch up to you.

The starting point for most commercial RTOS licenses is on the order of $10K for a non-royalty license, with full source code and full support. This represents a tremendous value—only costing a couple of months of a typical embedded firmware developer's salary. In addition, it's a one-time license purchase and the license can be used forever. That developer costs you $5K to $10K each and every month.

11. An RTOS is overkill for my application.

While it's hard to precisely state the criteria for an RTOS, when the total memory (ROM and RAM combined) of a device is less than 16 kB, there's a good chance that using an RTOS is overkill. Such devices typically have a dedicated purpose and most often use an 8- or 16-bit microprocessor. Once device firmware exceeds 32 kB of total memory and/or utilizes a communication protocol (like all IoT devices) or GUI, it will almost certainly need an RTOS.

Even a small 32-kB, dedicated purpose non-IoT device could benefit by using an RTOS, simply to isolate foreground and background processing into two separate threads. This configuration would only cost an additional 3 kB in total memory for the RTOS, while making the device firmware much simpler and much easier to enhance in the future.

With the rapid growth of the IoT and the new devices being developed to exploit it, using an RTOS in the near future seemingly becomes more and more likely.

References:

The Benefits of RTOSes in the Embedded IoT

Platforms for High-End Embedded Software Development

Source URL: http://electronicdesign.com/embedded/11-myths-about-using-rtos-iot-devices