Real-Time Operating System (RTOS)

Best Practices Guide

By: Jacob Beningo

Beningo Embedded Group

www.beningo.com

Sign-up for our monthly embedded systems newsletter at
http://www.beningo.com/newsletter-wp001
Introduction

Developing real-time software in a multi-tasking environment is a paradigm shift in thinking compared to the traditional super loop, bare-metal software that many developers are familiar with. Bare-metal techniques can emulate multi-tasking but in order to fully benefit from pre-emption, improved maintainability, reuse and reduced software complexity, a Real-Time Operating System (RTOS) is required. This guide provides best practices and recommendations for how to design embedded software using a Real-Time Operating System and is designed to be generic to cover most RTOSes with the primary target being resource constrained, microcontroller based systems.

The primary areas that will be covered by this guide include:

- Task Management
- Memory Management
- Performance Management
- Synchronization and Task Recommendations
- Issues and Debugging
Task Management Best Practices

Tasks are semi-independent applications that are executed by an RTOS kernel and the foundational object that every developer works with. In order to maximize performance, maintainability and reuse, developers should consider the following best practices:

Best Practice: Statically Allocate Tasks and RTOS Objects

Methodology:

- Look for this option in the RTOS configuration file so that tasks and objects are compiled statically at compile time
- If the option to compile statically is not available, create tasks and allocated objects in the application initialization so that dynamic allocation appears to be static allocation from the applications perspective

Exceptions:

- Microcontrollers that are heavily RAM constrained may not have enough RAM to statically allocate all tasks and objects. In this case, statically allocate as much as possible and then dynamically allocate the rest.

Best Practice: Avoid Dynamic Task Priorities

Methodology:

- Don’t change task priorities during run-time. The added complexity outweighs the minor benefits that can result. Managing task priorities at run-time can result in incorrect priority values that cause difficult to track down bugs

Exceptions:

- The only time task priorities should be changed during run-time is through a mutexes priority inheritance mechanism. The changed priority is then handled by the RTOS kernel and not up to the application developer to manage.

Best Practice: Minimize the Task Priority Levels Available

Methodology:
• Many RTOSes can be configured to allow task priority ranges from 0 – 1023. In many RTOSes, memory usage and performance can be improved by limiting the priority levels to 0 – 31. Change and verify these settings in the RTOS configuration file.

Exceptions:

• How task priority levels are implemented is dependent upon how the RTOS is written. Verify the most efficient priority ranges in the RTOS datasheet.

**Best Practice: Enable Time Slicing Over Round Robin**

**Methodology:**

• When tasks have the same priority level, an RTOS can usually be configured in multiple ways to determine how it will be behave. The first option, is to execute the tasks in a Round Robin fashion, where each task runs to completion before the next task executes. Alternatively, the tasks can be time sliced such that each task gets the CPU for a predetermined amount of time. Time slicing ensures that each task gets time on the CPU and is the preferred option.

Exceptions:

• If the tasks that will execute at the same priority level execute so fast that they would complete within a few time slices then using Round Robin may be the preferred option.

**Best Practice: Change the Default Stack Size**

**Methodology:**

• Check the RTOS configuration file for the default stack size. Each RTOS will have a different value but in most cases a default stack size should be 1024 bytes to start.

Exceptions:

• The stack size for any task will be determined by what that task is doing. The default value 1024 is a good starting value but in some applications the default may need to be much smaller or much bigger.

**Best Practice: Avoid Excessive Stack Usage**

**Methodology:**
- Avoid stack heavy operations in a task such as calling recursive functions and allocating large local structures. These operations can lead to stack overflows which are difficult to detect, repeat and debug.

Exceptions:

- None. Large local allocations should be done dynamically or statically and not on the local task stack. Recursive functions should always be avoided in resource constrained, real-time embedded systems.

**Best Practice: Functions used in Multiple Tasks must be Re-entrant**

**Methodology:**

- Functions that are used in multiple tasks simultaneously must be re-entrant to prevent data corruption. There are several features that a re-entrant function must exhibit:
  - The function return address should be stored on the stack and not in a register
  - The function cannot use global or static variables that are altered during the function execution

Exceptions:

- None

**Memory Management Best Practices**

The biggest excuse that bare-metal developers use to avoid using a RTOS is that they use too much memory and contain too much overhead. Improper memory management can certainly result in performance and memory footprint issues but a properly designed and architected application will not have any issues. Here are best practices developers can use to optimize their memory footprint and Below are a few tips for developers on how to minimize memory footprint and improve throughput.

**Best Practice: Create the Design Up-Front**

**Methodology:**

- Design the application before writing a single line of code. This will developers to optimize the number of tasks, semaphores, mutexes, message queues and other objects that are used in the application.
- Using too many RTOS objects can dramatically increase the RAM usage when it is not necessary.

Exceptions:
• None

Best Practice: Perform a Worst-Case Stack Analysis

Methodology:

• Each task will require a different stack size depending on what the task does. Sizing the task stack is not a one size fit all endeavor. Optimizing the stack size can dramatically improve memory usage. Many methods can be used such as:
  o Manual calculation
  o Static code analysis
  o Experimentation under load using watermarks and RTOS aware debuggers

Exceptions:

• A worst-case analysis should always be performed on system tasks to properly size the task. Developers will want to avoid overflowing the stack which can cause a system crash.

Best Practice: Minimize RTOS Objects

Methodology:

• Every task, semaphore, mutex, message queue, event flag and other RTOS object has a control block that is allocated to manage the resource. Each control block uses some RAM and in the case of tasks, stack space is also allocated. In many cases RTOS objects are overused, especially semaphores and developers should stop and decide if that object is necessary or if an alternative mechanism exists to do the same thing.

Exceptions:

• Developers shouldn’t minimize just to minimize. Applications should be optimally designed so that they are reusable and maintainable. These important features should not be overlooked in an exercise to simply minimize memory usage.

Best Practice: Use Memory Block Pools if Available

Methodology:
- Memory block pools are fast and efficient for allocating objects dynamically. They are deterministic and will not fragment unlike the heap or memory byte pools. If they are available in the RTOS, consider using them for dynamic memory allocation.

Exceptions:

- None.

**Best Practice: Minimize Creating and Destroying Objects in Run-Time**

**Methodology:**

- Dynamically allocating objects during run-time can result in heap fragmentation and memory being unavailable when needed. In order to prevent memory allocation issues, developers should:
  - Avoid creating and destroying objects at run-time
  - Use memory block pools if they do need to allocate and destroy objects

Exceptions:

- In very resource constrained devices, it may not be possible to avoid creating and destroying objects as they are needed. In these cases, it is perfectly acceptable to create and destroy objects but extra care should be taken to ensure that the memory needs are not exceeded or that fragmentation occurs.

**Best Practice: Use Block Pools over Byte Pools**

**Methodology:**

- RTOSes that include dynamic memory allocation will often provide two different options; byte and block pools. Byte pools behave just like the heap. They are prone to fragmentation and in some cases may not be deterministic. Block pools are deterministic and cannot fragment which is why they are preferred over the heap or byte pools.

Exceptions:

- None.
Performance Best Practices

Performance issues are the largest complaint that bare-metal developers site for not wanting to use an RTOS. In most instances, the performance issues are greatly exaggerated since the RTOS kernel will typically use 2 – 4% of the CPU utilization. In circumstances where developers are pushing the red line, these best practices can help improve their RTOS performance.

**Best Practice: Use Event Flags to Synchronize Tasks**

**Methodology:**

- Event flags typically use less memory and execute faster than semaphores. For this reason, evaluate the application design and determine if semaphores are being overused and if an Event Flag can be used instead. Make sure that you perform a test and read the RTOS documentation to verify that the event flags will be faster than a semaphore.

**Exceptions:**

- Using semaphores in many cases makes sense. Event Flags are great to signal when an event within the system has occurred such as sensor reading complete or system is over temperature as a few examples. Don’t blindly replace semaphores with event flags.

**Best Practice: Pass Data through a Message Queue by Pointer**

**Methodology:**

- Passing large amounts of data through a message queue starts to become inefficient the larger the data becomes. The reason is that the message queue needs to be sized so that it can copy the data into the queue. This copy mechanism can decrease performance and increase memory usage.

**Exceptions:**

- Passing a pointer to the data is great but that also means that the data being pointed to cannot change until the message data is consumed by the receiving task. If the data is updated before that happens then the task will miss some data. In this case, the performance hit may be worth the added synchronization needs.

**Best Practice: Minimize the Number of Tasks in the Application**
Methodology:

- One advantage to using an RTOS is that the application can be broken up into semi-independent programs that are more easily maintainable and reused; However, too many tasks can decrease the kernel throughput and dramatically increase memory usage. Therefore, developers should use the right number of tasks for the application and not break the application code up unnecessarily.

Exceptions:

- None.

Synchronization and Task Communication Recommendations

The synchronization tools available to developers within an RTOS offer many opportunities to accidentally use the wrong tool for the job. Developers can sometimes become confused as to whether they should be using a mutex or a semaphore or an event flag. In order to keep track where each synchronization tool should be used, below is a modified summary table which is an excerpt from “Real-Time Embedded Multithreading using ThreadX®” which shows the recommended resource to use (far left column) with the synchronization need (top row).

<table>
<thead>
<tr>
<th></th>
<th>Thread Synchronization</th>
<th>Event Notification</th>
<th>Mutual Exclusion</th>
<th>Inter-Thread Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex</td>
<td>Not Recommended</td>
<td>Not Recommended</td>
<td>Preferred</td>
<td>Not Recommended</td>
</tr>
<tr>
<td>Counting Semaphore</td>
<td>OK – better for one event</td>
<td>Preferred</td>
<td>OK</td>
<td>Not Recommended</td>
</tr>
<tr>
<td>Event Flags Group</td>
<td>Preferred</td>
<td>OK</td>
<td>Not Recommended</td>
<td>Not Recommended</td>
</tr>
<tr>
<td>Message Queue</td>
<td>OK</td>
<td>OK</td>
<td>Not Recommended</td>
<td>Preferred</td>
</tr>
</tbody>
</table>
The question that I get asked the most is “What is the difference between a mutex and a semaphore? Aren’t mutexes the same thing as a binary semaphore?”. The answer is that there are quite a few differences and mutexes are not the same thing as a binary semaphore. A good reference comparing the differences can again be found in “Real-Time Embedded Multithreading using ThreadX®” which can be found below:

<table>
<thead>
<tr>
<th>Speed</th>
<th>Semaphore</th>
<th>Mutex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>Typically faster and uses less memory</td>
<td>Slower and larger memory footprint</td>
</tr>
<tr>
<td>Thread Ownership</td>
<td>Ownership has no meaning to a semaphore</td>
<td>A single task can own a mutex</td>
</tr>
<tr>
<td>Priority Inheritance</td>
<td>Does not have this feature</td>
<td>Can be used with a mutex</td>
</tr>
<tr>
<td>Mutual Exclusion</td>
<td>Can be done with a binary semaphore but not recommended</td>
<td>This is the primary purpose of a mutex</td>
</tr>
<tr>
<td>Inter-Thread Communication</td>
<td>It is okay to use a semaphore for this purpose</td>
<td>Should NOT use a mutex in this manner</td>
</tr>
<tr>
<td>Event Notification</td>
<td>It is okay to use a semaphore for this purpose</td>
<td>Should NOT use a mutex in this manner</td>
</tr>
</tbody>
</table>

### Avoiding RTOS Issues and Debugging

Using an RTOS does not guarantee that an embedded system won’t have any issues. There is plenty that can go wrong if a developer is not careful. Below is a list of common RTOS issues and ways developers can avoid them.
Detecting issues in an RTOS can be challenging. Microcontrollers execute millions of instructions per second which can leave a developer scratching their head and guessing at what is really happening in their system. The following best practices can help developers minimize and discover bugs.

**Best Practice: Use assertions**

**Methodology:**

- The assert macro is included in the C standard library in assert.h. Assertions are used to test an assumption at a given point in an application and if the assumption tests false, there is a bug in the software.
- Assertions should be used to
  - Verify function preconditions
  - Verify function inputs
  - Verify function outputs
  - For bug detection only NOT error handling
Exceptions:

- Assertions should be used.
- Assertions can be disabled for production only if they are disabled before testing and verification. Assertions impact run-time performance due to the clock cycles necessary to evaluate an expression.

**Best Practice: Use Trace Tools**

**Methodology:**

- Trace tools provide an insight into the RTOS events that are occurring in the application such as when a task is ready to execute, when it starts executing, when it is blocked or pre-empted along with when synchronization objects used. A trace tool gives a developer visual insights into how the application is executing. Example tools that could be used include:
  - Percepio Tracealyzer
  - Segger System Viewer
  - Express Logix TraceX

Exceptions:

- None

**Best Practice: Enable Stack Monitoring**

**Methodology:**

- Stack monitoring will create a watermark in each task stack and monitor each task to determine how close to overflow the stack becomes. If an overflow occurs, an exception can be triggered that will give the software an opportunity to recover or at least make a developer aware that an issue exists.

Exceptions:

- In a heavily performance constrained system, the little bit of overhead associated with the stack monitor might affect the real-time performance. This is a rare situation but something to be aware of.

**Best Practice: Use Data Watches**

**Methodology:**
• A data watch sets up an internal comparator in the debug hardware that monitors a desired memory location for the data to change are be accessed in a specific manner. Data watches can be used to quickly detect issues with memory corruption.

Exceptions:

• None

**Best Practice: Understand your CPU Faults**

Methodology:

• RTOS applications can throw some of the most interesting and cryptic errors known to embedded software development. In order to quickly identify these issues, developers need to make sure that they understand their CPU error and exceptions and how to monitor them in their development environments.

Exceptions:

• None

**Going Further**

• Real-time Embedded Workshop [www.beningo.com](http://www.beningo.com)

**Recommended Resources**

Additional resources, templates and Jacob’s monthly embedded software newsletter can be found at [www.beningo.com](http://www.beningo.com). Check out his workshops at [http://www.beningo.com/training/](http://www.beningo.com/training/)

Click the social media link below to follow Jacob and get more tips and tricks:
Biography

Jacob Beningo, CSDP. Jacob Beningo is an embedded software consultant who currently works with clients in more than a dozen countries to dramatically transform their businesses by improving product quality, cost and time to market. He has published more than 200 articles on embedded software development techniques, is a sought-after speaker and technical trainer and holds three degrees which include a Masters of Engineering from the University of Michigan. Feel free to contact him at jacob@beningo.com or at his website www.beningo.com.

Testimonials

“Jacob Beningo conducted a 5-session hands-on class on the fundamentals of microcontrollers for EETimes University, that was sponsored by STMicroelectronics. While the “fundamentals” approach was familiar, we added the unique hands-on wrinkle and Jacob proved to be an excellent instructor. His success in leading the class was amply demonstrated in the outstanding participant engagement during the sessions and in their comments and feedback afterwards.”

-- Michael Markowitz, Director Technical Media Relations, STMicroelectronics

"Jacob was everything we could hope for in an embedded systems consultant. His understanding of the entire design process from requirements gathering, system architecting, implementation to testing and production was of great value to us. Jacob's layered software architecture design allowed us to quickly adapt to changing customer requirements saving both time and cost. He is easy to work with and has a knack for taking a complex technical topic and explaining it in easy to understand terms. I would recommend him and look forward to working with him on my next project."

-- Larry Roy, Vice President of Development, Embedded Logix

"Designing a robust and functional system starts with a good architecture design. Jacob worked with us to improve our embedded software architecture designs and helped us to implement a layered software architecture with clear and defined API's. His method for documenting software and adhering to software standards was phenomenal. We recommend and look forward to working with Jacob on future embedded software projects."

-- Ron Shelby, C&S Engineering