White paper

Why RTOS?

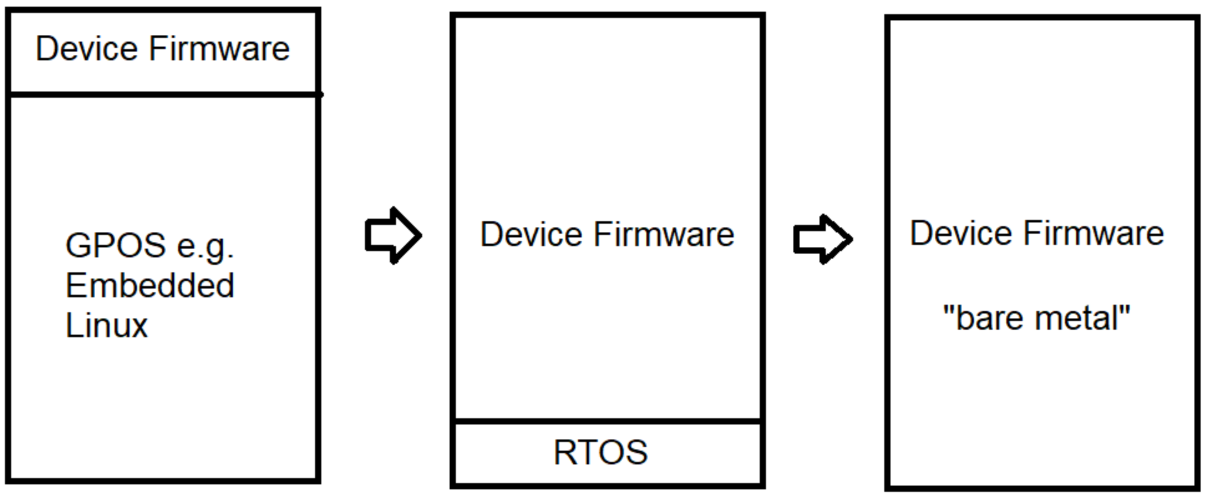
By Bill Lamie and Yuxin Zhou

Published January 25, 2023

Why RTOS?

By Bill Lamie and Yuxin Zhou, PX5

RTOS stands for Real-Time Operating System. An RTOS manages the processor's resources, including memory, as well as the flow of execution. Without an RTOS, there are two common alternatives: General Purpose Operating System (GPOS) and no operating system at, which is often referred to as "bare metal." The following shows the spectrum (from large-to-small) of embedded run-time solutions:



An RTOS differs from a General Purpose Operating System (GPOS) in that it is principally designed for resource constrained devices. GPOSes like Embedded Linux are primarily designed to offer a rich set of services that necessarily require significant amounts of memory (typically > 4MB) and processing power (typically > 1GHz). GPOSes also require a hardware Memory Management Unit (MMU) to support virtual addressing. Finally, GPOSes are typically not designed for hard real-time requirements. In sharp contrast, an RTOS is capable of operating in memory constrained environments – in some cases less than 32KB of memory. In addition, an RTOS can efficiently operate on microcontrollers running at less than 48MHz. Perhaps even more importantly, an RTOS is designed to meet hard real-time requirements, typically in the microsecond arena. The following is a comparison table to summarize some the differences between RTOS and GPOS environments:

	RTOS	GPOS
Minimal Memory	< 32KB	> 4MB
Minimal Clock Frequency	< 40MHz	> 1GHz
Hard Real-Time	Yes	No
MPU Support	Yes	Yes
MCU Support	Yes	No
MMU Required	No	Yes

Although GPOSes like Embedded Linux are not practical for many embedded microcontrollers, they may be good choices for larger MPU-based designs that require rich OS services. For resource constrained MCU-based designs and for MPU designs that have hard real-time requirements, an RTOS is generally a better solution.

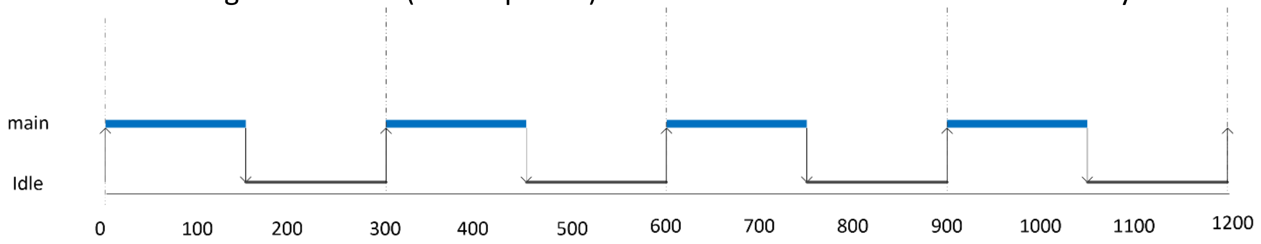
Even though an RTOS is often a good fit for resource constrained embedded devices, not all embedded devices benefit from an RTOS. The overhead of an RTOS isn't appropriate for some very small and singular purpose applications. These environments are typically referred to as "bare metal" implementations. This begs the question: When is it beneficial to use an RTOS?

A typical embedded application runs periodic tasks to process input from peripherals (such as sensors or communication devices) and generates output (sends data through communication devices or controls an actuator). For this discussion, let's assume an example device that performs machine learning tasks on images taken with an attached camera every 300ms. The following pseudo code accomplishes this task:

```
int main(…)
{
    while(1)
    {
       /* Get the time before the machine learning task. */
       time_before = get_time();

       /* Perform the task. */
       perform_machine_learning();

       /* Get time after the task. */
       time_after = get_time();
```

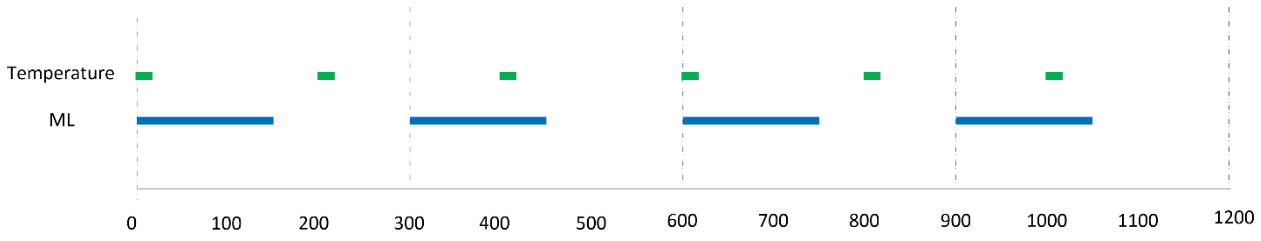
```
        /* Compute the amount of time spent on the task. */
        elapsed_time = time_before – time_after;

        /* Sleep for the rest of the 300ms window. */
        millisecond_sleep(300 – elapsed_time);
    }
    ......
}
```

The following timing diagram illustrates how the system behaves. Note the main thread performs the machine learning task every 300ms. When the machine learning task completes, the main thread goes into idle (or low power) mode for the remainder of the 300ms cycle.

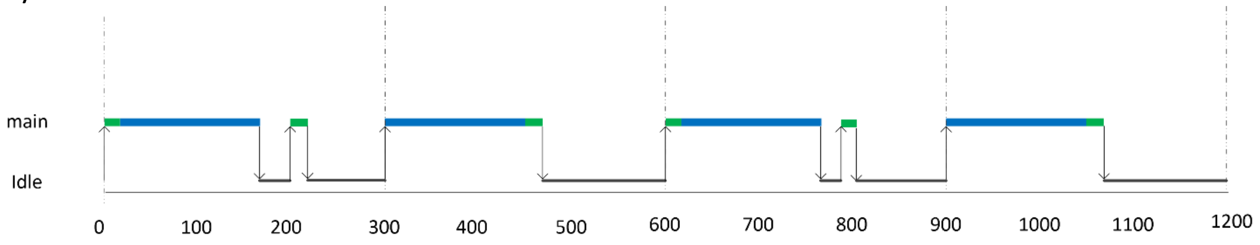


In this example, a simple "bare metal" control loop might be warranted, thereby eliminating the memory and processing cycles required by a more robust RTOS solution. However, it's important to note that all the responsibility for allocating processor cycles and meeting real-time requirements squarely on the application code.

Most embedded systems today handle more than one periodic task. Let's examine what it looks like as our example device now needs to process two periodic tasks. Suppose our device also needs to read temperature data every 200ms in addition to the existing 300ms machine learning task. Let's further assume that it is important to read the temperature at fixed intervals. That means the temperature reading has higher priority over the machine learning task. The system timing of our enhanced example device now looks like:



This timing diagram shows that some of the events occur concurrently. Specifically, the temperature readings at 0, 400ms, 600ms and 1000ms collide with the machine learning task execution. Understanding the real-time requirement, the developer might choose to schedule the temperature task before the machine learning task at 0ms and 600ms. However, this still leaves a problem for how to handle the temperature task at 400ms and 1000ms, which now must run after the machine learning task, since preemption is not supported. Of course, this introduces delays (or jitter) in the temperature task processing. If the real-time requirement doesn't allow this amount of jitter, the system fails. The timing diagram below illustrates this system behavior.



Recognizing that the jitter is caused by allowing the low priority task to run to completion, one could break the machine learning task into multiple segments, allowing the temperature task to execute between each segment, therefore reducing jitter. This approach increases the complexity of the system, making it more difficult to maintain. However, a bigger drawback is that it doesn't scale well as the number of periodic events increases with more complex timing requirements.

In addition to the periodic events, an embedded system also handles asynchronous events, typically generated by external interrupts. Adding to our example device, let's assume when a sensor detects a dangerous high-pressure situation, the program invokes a safety task to open a relieve valve. For the sake of this example, let's assume the system must respond to such an event within 30ms to prevent damage from the pressure build-up. Looking at the timing diagram above, we know that the program cannot wait for the machine learning process to finish before handling such critical event, as it will miss the 30ms deadline. To accomplish this requirement, one solution might be to further break down any long running tasks (like the machine learning task) into smaller segments, then building a state machine to allow control to temporarily leave the running task to handle the safety task (and temperature task). When the higher-priority tasks are complete, the machine learning task is resumed using the state machine to get back to the unfinished processing. Although this solution could meet the real-time requirements as stated, it is clear that the solution is becoming quite complicated, possibly inefficient, difficult to validate (as various logic units are intertwined), hard to maintain, and becomes exponentially more complex as each new requirement is added.

Another difficult issue to overcome with a simple control loop design is waiting for peripheral device I/O (blocking). Suppose *perform_machine_learning* is controlling an image sensor via peripheral I/O operations and must wait for the peripheral I/O device to complete its operation. Being blocked by IO in the machine learning task, could be adversely affect the higher priority temperature and safety task – especially if the peripheral I/O operation is not deterministic. In this situation, the application developer would likely be forced to add another state in the state machine inside of *perform_machine_learning* such that it wouldn't ever wait, instead setting up a state machine such that it could "find its way back" to the peripheral I/O device operation to check for completion. This additional state machine processing necessarily adds complexity and overhead.

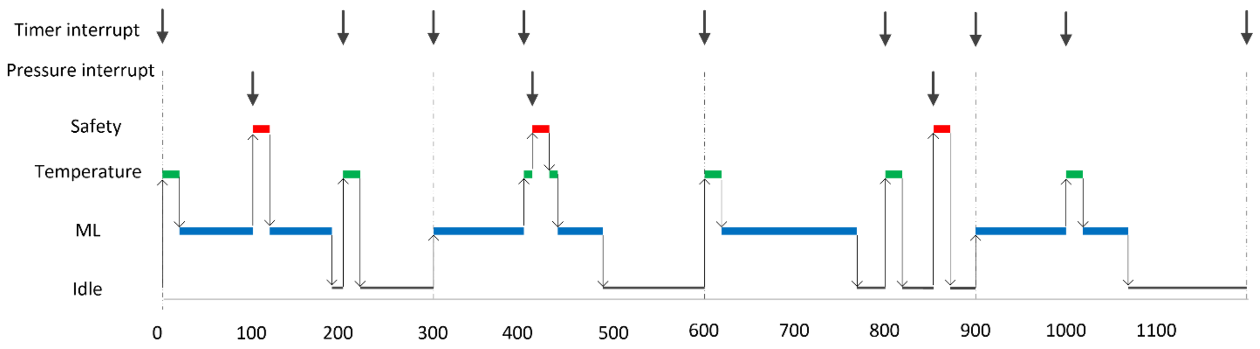
From the developer team point of view, there is another drawback developing complex system using a simple control loop: The project would require every developer to have precise knowledge of the real-time processing of every component. Not only does this become exponentially more difficult as the complexity and real-time requirements increase, it also doesn't scale well as more developers are added to the project.

Let’s now examine what the same example device looks like using an RTOS. With the aid of an RTOS, the example device can be restructured using preemptive programing. The program can be partitioned into 3 threads: the machine learning thread, the higher priority temperature thread, and the highest-priority pressure-sensor thread. The following pseudo code shows the implementation of these 3 threads. Note that with RTOS support, the periodic tasks rely on timer interrupts to wake them up.

```
int main(…)
{
   pthread_create(&thread_0, NULL, machine_learning_entry, NULL);
   pthread_create(&thread_1, NULL, temperature_entry, NULL);
   pthread_create(&thread_2, NULL, safety_entry, NULL);

}

void * machine_learning_entry(void *argument)
{
   while(1)
   {
      wait_for_300ms_interrupt();
      perform_machine_learning();
   }
}

void * temperature_entry(void *argument)
{
   while(1)
   {
      Wait_for_200ms_interrupt();
      read_temperature();
   }

}

void * safety_entry(void *argument)
{
   while(1)
```

```
    {
      wait_for_pressure_sensor_interrupt();
      open_relief_valve();
    }
}
```

The execution looks like:



Using an RTOS, all the high priority events meet their timing requirements. A complex system can be easily broken down into multiple threads, each of which can implement its own simple control loop. The priority of each thread determines when each thread executes: when a higher- priority thread is ready, it preempts the executing lower-priority thread, thereby guaranteeing the higher-priority deadline is met. The beauty of an RTOS-based design is that boundless new functionality can be added without adversely affecting the real-time processing in the safety or temperature threads, assuming, of course, that they have higher priority.

In summary, an RTOS is generally a better choice than a GPOS for resource constrained devices or devices that have hard real-time requirements. At the other end of the spectrum, an RTOS is generally better than a "bare metal" simple control loop when the device firmware has hard real-time requirements and/or sufficient complexity (> 32K, network connectivity, device I/O, etc.). Even in situations where the application could "get by" with a simple control loop, there is still the benefit of using an RTOS to help "future proof" the device firmware. Here are some advantages of using an RTOS in bullet form:

- Generally better than GPOS for resource constrained or hard real-time environments
- Enhances application real-time responsiveness
- Reduces complexity and makes development easier
- Easier to divide an application into more manageable pieces
- Enables more features and project developers
- Possible reduction of overhead via elimination of polling and state machines
- Achieve concurrency by enabling other processing while waiting for blocking I/O
- Enable true parallel processing in symmetric multiprocessing RTOS environments

Enhance • Simplify • Unite

11440 West Bernardo Court • Suite 300
San Diego, CA 92127, USA

Phone: +1 (858) 753-1715
Email: info@px5rtos.com
Website: px5rtos.com

© PX5 • All Rights Reserved