By Mats Pettersson, IAR Systems

Basic Concepts for Real Time Operating Systems

This article will explain some Real Time Operating Systems (RTOS) basics. The material presented here is not intended as a complete coverage of different RTOS and their features. For the purposes of simplification, I will briefly cover the most important features of a representative RTOS.

After we explore what constitutes an RTOS and why we might want to use one, I'll explain each of the basic components of a typical RTOS and show how these building blocks are integrated into the system.

For the few code examples in this article we will use ThreadX from Express Logic.

Thread-oriented design

Designing applications for an embedded application is almost always challenging, to say the least. One way to decrease the complexity of your application is to use a thread-oriented design and divide a project into more manageable pieces (or threads). Each thread is then responsible for some part of the application. With such a system you would like to be able to specify that some thread is more important than others. That is, some threads have real-time requirements. They have to respond quickly and correctly. If your system employs a professional RTOS, features that prioritize threads are already part of the package. In addition to thread prioritization, a clean and well-tested API is included that eases communicate between different threads.

So if we to use a professional RTOS we will have the tools to:

- Ensure that time-critical parts of the code execute within their real-time constraints.
  Perhaps equally importantly, the real-time behavior of the high-priority threads is not affected by the number or processing of less important, lower-priority threads.

- Make complex applications easier to develop and maintain.
  It's easier to develop and maintain smaller threads, than to have to deal with the entire application as a whole. In addition, changes to the processing of lower priority threads do not affect the real-time processing of higher-priority threads.

- Distribute different parts of the application among several developers.
  Each developer can be responsible for one or more threads within the application and a clean Application Programming Interface (API) will be available for communication between the different modules/threads as they are developed.

Applications can be divided into different threads with or without the use of an RTOS. For example, one thread can be responsible for reading the keyboard, another for checking temperature, a third for printing messages on a LCD screen, and so on. With an RTOS you not only get the tool to create threads, but also tools to communicate between the threads, and tools to ensure that threads that are time critical are executed within their real-time constraints. Since the interface between the different threads becomes very clean when using an RTOS you will save time both in development and in maintenance of the application.

How does an RTOS work?

The core of an RTOS is known as the **kernel**. An API is provided to allow access to the kernel for the creation of threads, among other things. A **thread** is like a function that has its own stack, and a Thread

Control Block (**TCB**). In addition to the stack, which is private to a thread, each thread control block holds information about the state of that thread.
The kernel also contains a **scheduler**. The scheduler is responsible for executing threads in accordance with a scheduling mechanism. The main difference among schedulers is how they distribute execution time among the various threads they are managing. Priority-based, preemptive scheduling is the most popular and prevalent thread scheduling algorithm for embedded RTOSes. Typically, threads of the same priority execute in a round-robin fashion.

Most kernels also utilize a system tick interrupt, with a typical frequency of 10ms. Without a system tick in the RTOS, basic scheduling is still available, but time-related services are not. Such time-related services include: software timers, thread sleep API calls, thread time-slicing, and timeouts for API calls.

The system tick interrupt can be implemented with one of the hardware timers in the embedded chip. Most RTOS have the ability or extension to reprogram the timer interrupt frequency dynamically such that the system can sleep until the next timer expiration or external event. For example, if you have an energy sensitive application you might not want to run the system tick handler every 10ms if not necessary. Suppose for example the application is idle and the next timer expiration is 1000ms away. In this case, the timer can be reprogrammed to 1000ms and the application can enter low-power mode. Once in this mode, the processor will sleep until either another external event occurs or the 1000ms timer expires. In either case, when the processor resumes execution the RTOS adjust the internal time according to how much time has elapsed and normal RTOS and application processing is resumed. This way, the processor only executes when the application has something to do. During idle periods the processor can sleep and save power.

There will be further discussion about scheduling algorithms and system ticks later in this article.

Think thread...

Perhaps the best way to get started with an RTOS application is to think about how the application can be divided into distinct threads. For example, a somewhat simple engine input control application could be divided into the following threads:

- Engine temperature
- Oil pressure
- Rotation per minute (RPM)
- User input

These modules can then be set up as threads, or can be divided into sub-threads. For example:

- Engine temperature
  - Read engine temperature
  - Update LCD with current temperature
- Oil pressure
  - Read current oil pressure
  - Conduct emergency engine shutdown
- RPM
  - Read RPM
  - Update LCD with current RPM
- User input
  - Get gas pedal angle
  - Get current gear

This division into sub-threads can continue until work can be handled by a single thread.

RTOS components

Let's see what features an RTOS has to offer and how each of these features might come in handy in different applications.

```
void ThreadSendData( void )
{
  while (1)
  {
  // Send the data…
  }
}
```

Threads

Threads are like functions, each with its own stack and thread control block (TCB). Unlike most functions, however, a thread is almost always an infinite loop. That is, once it has been created, it will never exit.

A thread is always in one of several **states**. A thread can be ready to be executed, that is, in the **READY** state. Or the thread may be suspended (pending), that is, the thread is waiting for something to happen before it goes into the READY state. This is called the **WAITING** state.

Here is a short description of the states we will use in ThreadX.

State	Description
Executing	This is the currently-running thread.
Ready	This thread is ready to run
Suspended	This thread is waiting for something. This could be an event, a message or maybe for the RTOS clock to reach a specific value (delayed).
Completed	A thread in a completed state is a thread that has completed its processing and returned from its entry function. (A thread in a completed state cannot execute again.)
Terminated	A thread is in a terminated state because another thread or the thread itself called the tx_thread_terminate service. (A thread in a terminated state cannot execute again.)

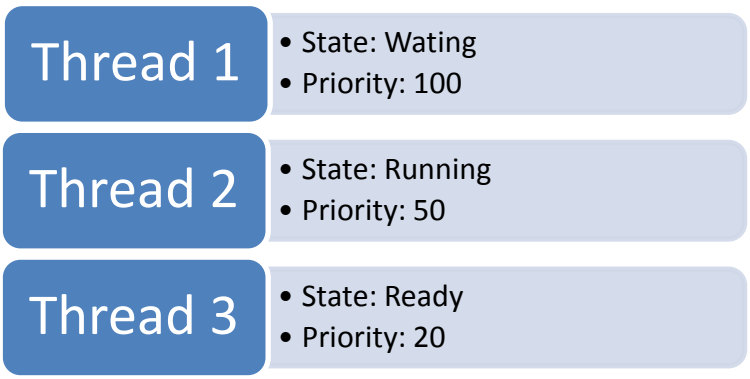
***Note**: different RTOS might have different names for each of these states*

Scheduler

There are two major types of scheduler from which you can choose:

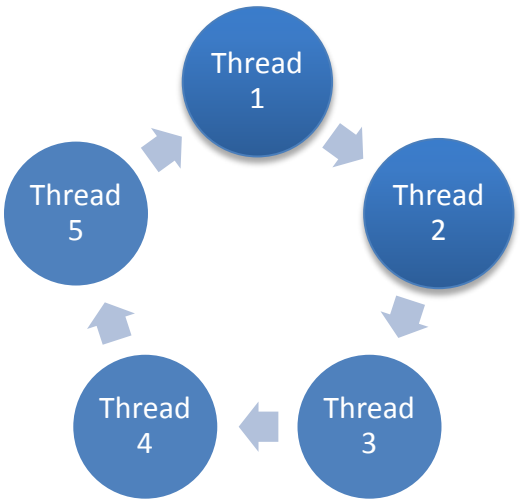
1. **Event-driven (Priority-Controlled Scheduling Algorithm)**
   Usually different threads have differing response requirement. For example, in an application that controls a motor, a keyboard and a display, the motor usually requires faster reaction time than the keyboard and display. This makes an event-driven scheduler a must.

   In event-driven systems, every thread is assigned a priority and the thread with the highest priority is executed. The order of execution depends on this priority. The rule is very simple: **The scheduler activates the thread that has the highest priority of all threads that are ready to run.**



2. Time-sharing

The most common time-sharing algorithm is called **Round-Robin**. With round-robin scheduling, the scheduler has a list of the threads that make up the system and it uses this list to check for the next thread that is ready to execute. If a thread is READY, that thread will execute. Associated with each thread is its 'time-slice'. This time-slice is the maximum time a thread can execute for each round the scheduler makes.



A typical priority-based, preemptive scheduler supports both **preemptive** and **non-preemptive** scheduling. In a **preemptive** situation, a higher-priority thread immediately interrupts (preempts) an executing, but lower priority thread. Threads of the same priority are scheduled in a **non-preemptive** fashion, where the executing thread will finish its execution before another thread of the same or lower-priority executes.

Assigning priorities

It is very important to assign correct priorities to your different threads. A lot of papers have been written about how this can be done in order to get the most out of your RTOS based application. We will not dive deep into this subject but just mention a couple of rules that are helpful.

1. Use as few priority levels as possible.
   Only assign different priorities when preemption is absolutely necessary. This will reduce the amount of context switches done in the system. And the fewer context switches done, the more time can be spent on executing application code.

2. Make sure that all critical timing constraints are met in your application.
   Depending on what type of application you have, this can be a tough one. One way of solving this is to use RMA (Rate Monotonic Algorithm). The ThreadX RTOS also provides a unique technology called preemption-threshold. This can be used to reduce context switches as well as help guarantee the execution of application threads. See:
   http://www.nxtbook.com/nxtbooks/cmp/esd0311/#/26

More information about assigning priorities can be found for example on the Internet. Below are links to two articles on this subject for you who like to read more about this.

- http://www.eetimes.com/design/embedded/4008216/Keeping-your-priorities-straight-Part-1--context-switching
- http://www.netrino.com/node/77

Thread communications

We also need to be able to communicate between the different threads in an RTOS. Communication can take the form of an event, a semaphore (flag), or it can be in the form of a message sent to another thread.

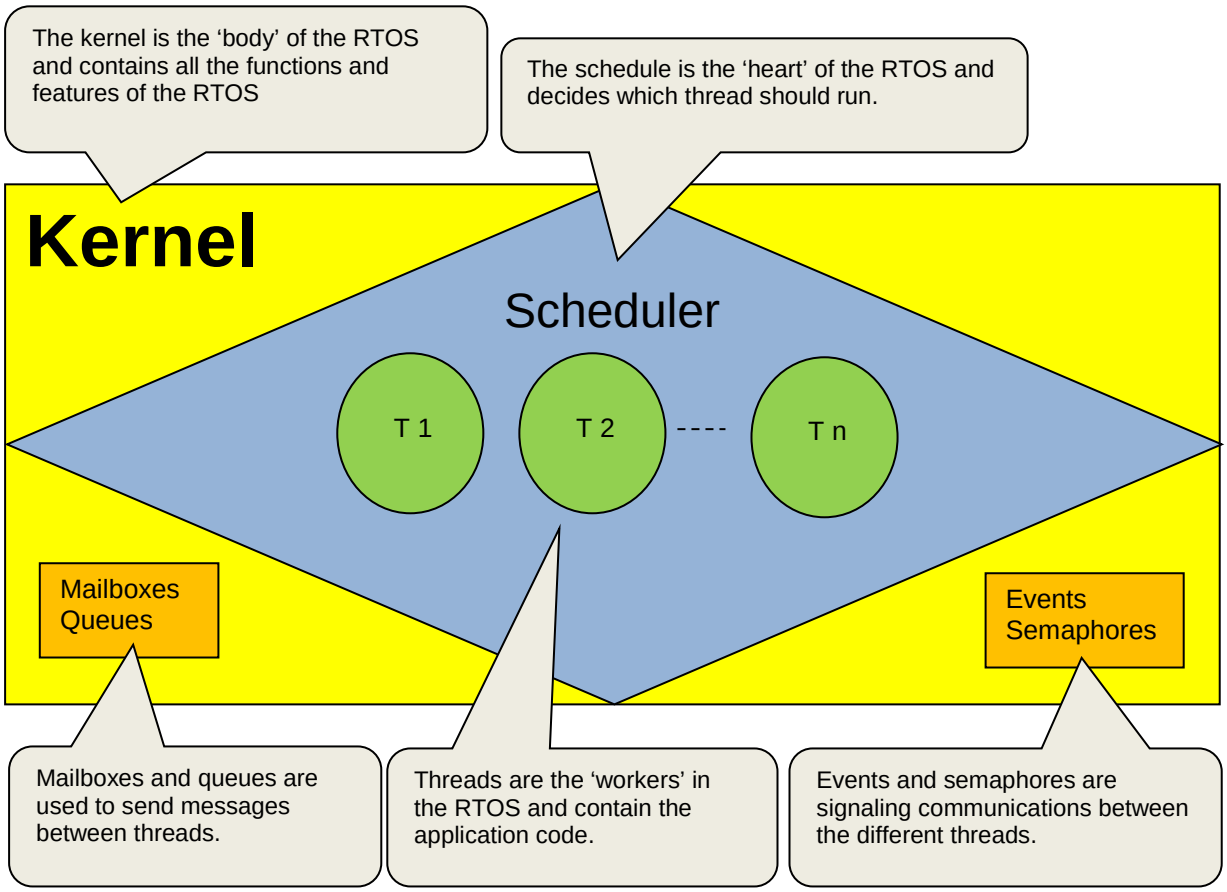
The most basic communication is via an **event**. An **event** is a way for a single thread to communicate with another thread. An interrupt service routine (ISR) can also send an event to a thread. Some RTOS can also send one (1) event to more than one thread.

Semaphores are usually used to protect shared resources, for example if there is more than one thread that needs to read/write to the same memory (variable). This is done so that a variable does not change through the actions of another thread while it is being addressed by the active thread. The principle is that you need to obtain a semaphore associated with a variable protected in this way before reading/writing to this memory. Once you have obtained the semaphore, no one else can read/write to this memory until you release the semaphore. This way you can ensure that only one thread at the time reads/writes to a memory location or variable.

Messages allow you to send data to one or more threads. These messages can be of almost any size and are usually implemented as a **mailbox** or a **queue**. The behavior for mailboxes and message queues varies from different RTOS vendors.

Putting it all together...

Let's recap what we have discussed so far.



Now then, let's see what we can do with the components we have so far. Imagine that we need to create an engine control application and we have a microcontroller and a LCD screen. We would also like to have the current time displayed on the LCD. We will reuse the engine control example, where we divided this application into different modules and threads. (We will ignore the "User input" to make the example clearer.)

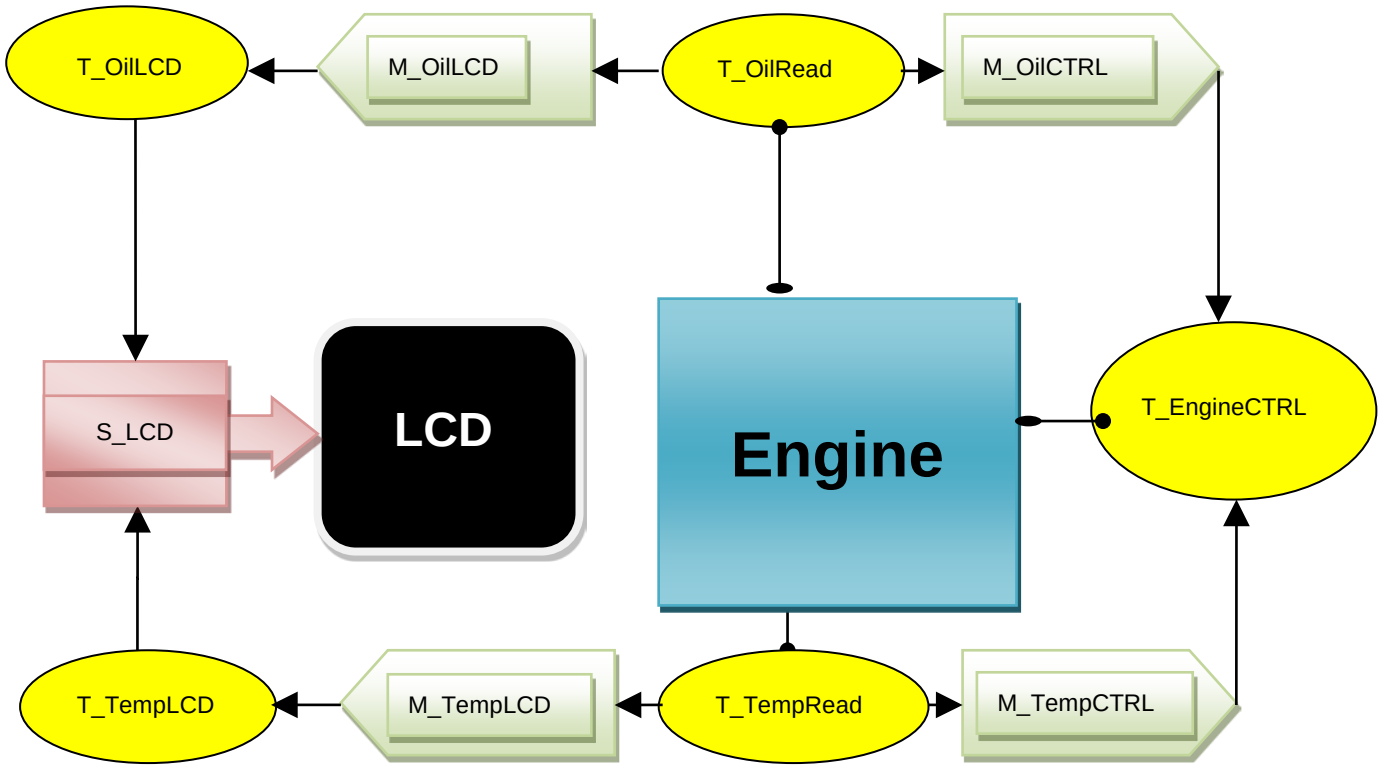
For this exercise, we will create the following threads:

1. // Read the temperature and oil pressure of the engine
   void T_OilRead(void) and **void T_TempRead(void)**

2. // Print the temperature and oil pressure on the LCD
   void T_OilLCD(void) and **void T_TempLCD(void)**

3. // Controls the engine
   void T_EngineCTRL(void)

Graphically, the system will look like this:



And the different symbols used are defined as follows…

For this system, we will need a semaphore to control writes to the LCD. We need this because we have two different threads that are writing to the LCD and if one of these threads was interrupted by the other, the output would most likely be corrupted.

We need the following signaling mechanisms.

- The mailboxes **M_TempLCD** and **M_OilLCD.**
  These mailboxes contain messages that will be printed on the LCD.

- The mailboxes **M_TempCTRL** and **M_OilCTRL.**
  These mailboxes contain messages that are sent to the engine control thread.

- The semaphore **S_LCD.**
  This semaphore will ensure that one and only one thread at a time prints to the LCD.

System ticks
We now have a system that will update the LCD with motor commands and show the current time on the LCD. There is, however, one very important thing missing and that is the RTOS tick function. As stated earlier, the kernel needs to gain control over the system every now and then to be able to, for example, switch threads according to the RTOS scheduler. This is normally done by calling a tick-handling API function driven by one of the microcontroller timers. Without system ticks, nothing will happen in the system.

Connection to the microcontroller hardware
The last piece in the puzzle is to connect all this to the hardware. In our case we will assume that we have access to a firmware library. We assume that we have the following firmware API functions:

- // Set text X,Y coordinate in characters
  void **GLCD_TextSetPos**(int X, int Y);

- // Set draw window XY coordinate in pixels
  void **GLCD_SetWindow**(int X_Left, int Y_Up, int X_Right, int Y_Down);

- // Function that reads the engine temperature
  char **Engine_ReadTemp**(void);

- // Function that reads the engine oil pressure
  char **Engine_ReadOilPressure**(void);

- // Function that controls the engine
  char **Engine_Control**(int CtrlValue);

Now we only have to put the pieces together to see if our system will work as intended. The easiest way to get started is, if available, to use a BSP from the RTOS vendor. For most RTOS's there are numerous of different BSP for various devices.

To build your application without a BSP you have to:

- Add compiler/assembler include paths to the build tool.
  We need to make sure that the compiler and assembler can find the header files used in our system.

- Add the generic RTOS library or source files.
  We need to add the kernel. The kernel can come either as source or as library files.

- Add the target-specific RTOS files.
  Many RTOS have a Board Support Package (**BSP**) for different boards/devices. In such a case, you only have to make sure that you include these target-specific files. In such a BSP you would, for example, have code that configures the clock so that the RTOS tick is configured correctly.

- Your application has to initialize the RTOS and create at least one thread before starting the RTOS. This is usually done in the main() function, but can be done anywhere in your application.

Assuming that we have added all of the files and have set the build options according to the RTOS specifications, we are ready to start creating threads in our application. But before we can create our threads, we need to define their Thread Control Blocks (TCB).

Below is an example of a definition of the TCB (Thread Control Block) for Express Logic ThreadX.

Express Logic ThreadX example

```
TX_THREAD TCBEngineCTRL;
TX_THREAD TCBOilLCD;
TX_THREAD TCBTempLCD;
TX_THREAD TCBOilRead;
TX_THREAD TCBTempRead;
```

Now, what remains is to create our main function from where we create the threads that we need when we start our application.

In ThreadX we create our initial threads in the API function ‘tx_application_define’. So in main the only thing we have to do is to call the API function ‘tx_kernel_enter’. ThreadX will then call tx_application_define() where we create our tasks and semaphores and message queues and similar that we might need from the start.

Most RTOS will allow you to create resources, as for example threads, dynamically as you run your application. But we will need to have at least one thread created as the scheduler takes control over our application.

When this is done we can start our RTOS and let the scheduler take care of the problem of making sure that the correct thread is executed at the right time.

Summary

This article explains the concepts of kernel, threads, messaging (like events and mailboxes), scheduler and the main pieces to put an RTOS application together. To do this in a real application will require more details, for example, how to create mailboxes and events.

The best way to get started, if you are new to using an RTOS, is to bring up and use one of the many examples that come from the different RTOS vendors. In the article above I have used Express Logic for the code snippets. Express Logic provides many example applications and has BSP’s for many different boards and devices.