RTOS Reference

Contents

Introduction

You have access to a multithreaded, preemptive, real-time operating system (RTOS) that implements thread priority. The RTOS also provides services to threads for communication, synchronization and coordination. The RTOS has been designed for use in a "hard" real-time system – i.e. where threads have to execute not only correctly but also in a timely fashion.

R.1 RTOS Services

The following list shows the services that the RTOS provides to applications.

Name	Service
`OS_Init()`	Initialize the RTOS
`OS_ISREnter()`	Signal ISR entry
`OS_ISRExit()`	Signal ISR exit
`OS_SemaphoreCreate()`	Create a semaphore
`OS_SemaphoreSignal()`	Signal semaphore
`OS_SemaphoreWait()`	Wait on semaphore
`OS_Start()`	Start multithreading
`OS_ThreadCreate()`	Create a thread
`OS_ThreadDelete()`	Delete a thread
`OS_TimeDelay()`	Delay a thread for *n* system ticks
`OS_TimeGet()`	Get the system time, in ticks
`OS_TimeSet()`	Set the system time

R.2 RTOS Structure

This section describes some of the structural aspects of the RTOS:

- how the RTOS handles access to critical sections of code
- how the RTOS knows about your threads
- how threads are scheduled
- how to write Interrupt Service Routines (ISRs)
- what a clock tick is and how the RTOS handles it
- how to initialize the RTOS
- how to start multithreading

This section also describes the following application services:

- `OS_DisableInterrupts()` and `OS_EnableInterrupts()`
- `OS_Init()`
- `OS_Start()`
- `OS_ISREnter()` and `OS_ISRExit()`

R.2.1 Critical Sections

The RTOS, like all real-time kernels, needs to disable interrupts in order to access critical sections of code and to reenable interrupts when done. This allows the RTOS to protect critical code from being entered simultaneously from either multiple threads or ISRs. The interrupt disable time is one of the most important specifications of a RTOS because it affects the responsiveness of your system to real-time events. The RTOS defines two macros to disable and enable interrupts:

- `OS_EnterCritical()` and
- `OS_ExitCritical()`,

respectively.

R.2.2 Threads

A thread is typically an infinite loop function (2) as shown in Listing R.1. A thread looks just like any other C function containing a return type and an argument, but it never returns. The return type must always be declared **`void`** (1).

```
void YourThread(void* pData)      (1)
{
  for (;;)                        (2)
  {
    // User Code
    // Call one of the RTOS's services:
    OS_SemaphoreWait(...);
    ...
    // Call another of the RTOS's services:
    OS_TimeDelay(...);
    // User Code
    ...
  }
}
```

Listing R.1 – A thread is an infinite loop

Alternatively, the thread can delete itself upon completion as shown in Listing R.2. Note that the thread code is not actually deleted; the RTOS simply doesn't know about the thread anymore, so the thread code will not run. Also, if the thread calls `OS_ThreadDelete()`, the thread never returns.

```
void YourThread(void *pData)
{
  // User Code
  ...
  OS_ThreadDelelete(OS_PRIORITY_SELF);
}
```

Listing R.2 – A thread that deletes itself when done

The argument (1) is passed to your thread code when the thread first starts executing. Notice that the argument is a pointer to a **`void`**. This allows your application to pass just about any kind of data to your thread. The pointer is a "universal" vehicle used to pass your thread the address of a variable, a structure, or even the address of a function if necessary! It is possible to create many identical threads, all using the same function (or thread body). For example, you could have two serial ports that each are managed by their own thread. However, the thread code is actually identical. Instead of copying the code twice, you can create a thread that receives a pointer to a data structure that defines the serial port's parameters (baud rate, I/O port addresses, interrupt vector number, etc.) as an argument.

The RTOS manages up to 32 threads; however, the RTOS uses one thread (the `Idle()` thread) for system use. Therefore, you can have up to 31 application threads. Each thread of your application must be assigned a unique priority level from 0 to 31. The lower the priority number, the higher the priority of the thread. The RTOS always executes the highest priority thread ready to run. The thread priority number also serves as the thread identifier. The priority number (i.e., thread identifier) is used by some kernel services such as `OS_ThreadDelete()`.

In order for the RTOS to manage your thread, you must "create" a thread by passing its address along with other arguments to the function `OS_ThreadCreate()`.

R.2.3 Thread States

Figure R.1 shows the state transition diagram for the threads. At any given time, a thread can be in any one of four states.

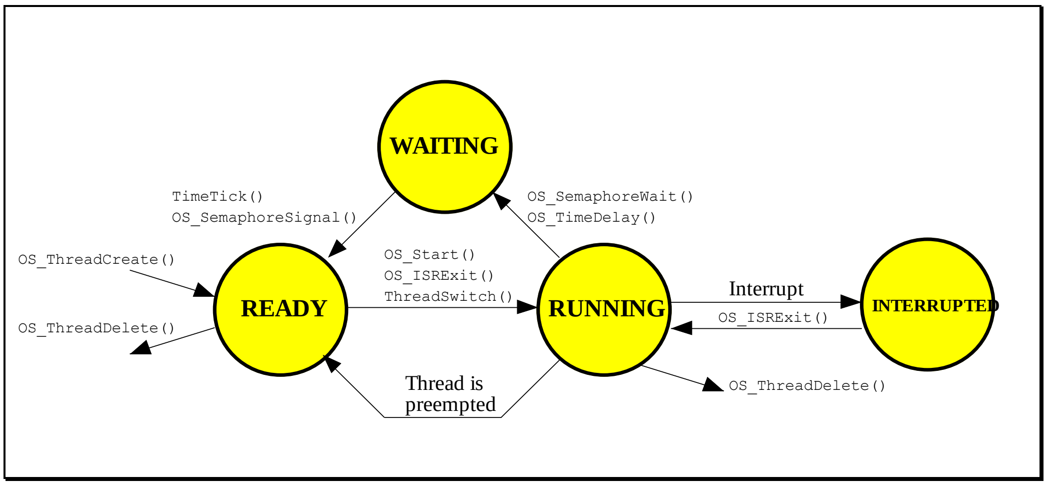

Figure R.1 – Thread states

A thread is made available to the RTOS by calling `OS_ThreadCreate()`. When a thread is created, it is made `READY` to run. Threads may be created before multithreading starts or dynamically by a running thread. If a thread created by another thread has a higher priority than its creator, the created thread is given control of the CPU immediately. A thread can return itself or another thread to the dormant state by calling `OS_ThreadDelete()`.

Multithreading is started by calling `OS_Start()`. `OS_Start()` runs the highest priority thread that is `READY` to run. This thread is thus placed in the `RUNNING` state. Only one thread can be running at any given time. A ready thread will not run until all higher priority threads are either placed in the wait state or are deleted.

The running thread may delay itself for a certain amount of time by calling `OS_TimeDelay()`. This thread is `WAITING` for some time to expire and the next highest priority thread that is ready to run is given control of the CPU immediately. The delayed thread is made ready to run by the RTOS when the desired time delay expires.

The running thread may also need to wait until an event occurs by calling OS_SemaphoreWait(). The thread is thus WAITING for the occurrence of the event. When a thread waits on an event, the next highest priority thread is given control of the CPU immediately. The thread is made ready when the event occurs. The occurrence of an event may be signalled by either another thread or an ISR.

A running thread can always be INTERRUPTED, unless the thread or the RTOS disables interrupts. When an interrupt occurs, execution of the thread is suspended and the ISR takes control of the CPU. The ISR may make one or more threads ready to run by signalling one or more events. In this case, before returning from the ISR, the RTOS determines if the interrupted thread is still the highest priority thread ready to run. If a higher priority thread is made ready to run by the ISR, the new highest priority thread is resumed; otherwise, the interrupted thread is resumed.

When all threads are waiting either for events or for time to expire, the RTOS executes the idle thread, Idle().

R.2.4 Thread Control Blocks

When a thread is created, it is assigned a Thread Control Block, TCB (Listing R.3). A thread control block is a data structure that is used by an RTOS to maintain the state of a thread when it is preempted. When the thread regains control of the CPU, the thread control block allows the thread to resume execution exactly where it left off. All TCBs reside in RAM. A TCB is initialized when a thread is created.

```
typedef struct tcb
{
  void* pStack;         // Pointer to current top of stack
  uint8_t priority;     // Thread priority (0 = highest)
  OS_STATE state;       // Thread state
  Uint32_t delay;       // Number of ticks to delay or timeout
  OS_ECB* pEvent;       // Pointer to event control block
  struct tcb* next;     // Pointer to next TCB in TCB list
  struct tcb* prev;     // Pointer to previous TCB in TCB list
} OS_TCB;
```

Listing R.3 – An RTOS thread control block

.pStack contains a pointer to the current top-of-stack for the thread. The dot (.) in front of the variable name indicates that it is part of a structure, and not a global variable. The RTOS allows each thread to have its own stack, but just as important, each stack can be any size.

.priority contains the thread priority. A high-priority thread has a low .priority value (i.e., the lower the number, the higher the actual priority).

.state contains the state of the thread. When .state is OS_STATE_READY, the thread is ready to run. Other values can be assigned to .state, and these values are described in OS.h.

.delay is used when a thread needs to be delayed for a certain number of clock ticks or a thread needs to wait for an event to occur with a timeout. In this case, this field contains the number of clock ticks the thread is allowed to wait for the event to occur. When this variable is 0, the thread is not delayed or has no timeout when waiting for an event.

.pEvent is a pointer to an event control block.

.next and .prev are used to doubly link TCBs. This chain of TCBs is used by the RTOS to update the .delay field for each thread. The TCB for each thread is linked when the thread is created, and the TCB is removed from the list when the thread is deleted. A doubly linked list permits an element in the chain to be quickly inserted or removed.

The maximum number of threads (OS_MAX_USER_THREADS) that an application can have is specified in OS.h and determines the number of TCBs allocated by the RTOS for your application. You can reduce the amount of RAM needed by setting OS_MAX_USER_THREADS to the actual number of threads needed in your application.

R.2.5 Thread Priority

Each thread is assigned a unique priority level between `0` and `OS_LOWEST_PRIORITY`, inclusive (see `OS.h`). Thread priority `OS_LOWEST_PRIORITY` is always assigned to the `Idle()` thread when the RTOS is initialized. Note that `OS_MAX_USER_THREADS` and `OS_LOWEST_PRIORITY` are unrelated. You can have only 5 threads in an application while still having 8 priority levels (if you set `OS_LOWEST_PRIORITY` to 7).

R.2.6 Thread Scheduling

The RTOS always executes the highest priority thread ready to run. The determination of which thread has the highest priority, and thus which thread will be next to run, is determined by the RTOS's scheduler.

R.2.7 Idle Thread

An RTOS always creates an idle thread that is executed when none of the other threads is ready to run. The idle thread `Idle()` is always set to the lowest priority, `OS_LOWEST_PRIORITY`. The idle thread can never be deleted by application software.

R.3 Interrupt Processing

The C code for an ISR is shown in Listing R.4.

```
void __attribute__ ((interrupt)) MyISR(void)
{
  OS_ISREnter();                              (1)
  // Clear interrupt flag                     (2)
  ...
  // Execute user code to service ISR         (3)
  ...
  OS_ISRExit();                               (4)
}                                             (5)
```

Listing R.4 – An ISR using the RTOS

On entry into the ISR, the RTOS needs to know that you are servicing an ISR, so you need to call `OS_ISREnter()` (1). Then, you should clear the interrupt flag (2); otherwise, your interrupt will be re-entered at the end of the ISR and your application will be trapped in an endless loop! The function `OS_ISREnter()` increments a counter to tell how many interrupts have been "nested".

Once the previous two steps have been accomplished, you can start servicing the interrupting device (3). This section is obviously application specific. The RTOS allows you to nest interrupts because it keeps track of nesting in `OS_ISREnter()`.

The conclusion of the ISR is marked by calling `OS_ISRExit()` (4), which decrements the interrupt nesting counter. When the nesting counter reaches `0`, all nested interrupts have completed and the RTOS needs to determine whether a higher priority thread has been awakened by the ISR (or any other nested ISR). If a higher priority thread is ready to run, the RTOS returns to the higher priority thread rather than to the interrupted thread. If the interrupted thread is still the most important thread to run, `OS_ISRExit()` returns to its caller (4). At that point an exception return instruction is executed (5).

The above description is further illustrated in Figure R.2. The interrupt is received (1) but is not recognized by the CPU, either because interrupts have been disabled by the RTOS or your application or because the CPU has not completed executing the current instruction. Once the CPU recognizes the interrupt (2), the CPU vectors to the ISR (3). Once this is done, your ISR notifies the RTOS by calling `OS_ISREnter()` (4). Your ISR code then gets to execute (5). Your ISR should do as little work as possible and defer most of the work to a thread. A thread is notified of the ISR by calling `OS_SemaphoreSignal()`. The receiving thread may or may not be waiting for the semaphore when the ISR occurs and the signal is made. Once the user ISR code has completed, you need to call `OS_ISRExit()` (6). As can be seen from the timing diagram, `OS_ISRExit()` takes less time to return to the interrupted thread when there is no higher priority thread (HPT) readied by the ISR. Furthermore, in this case, an exception return instruction is executed (7). If the ISR makes a higher priority thread ready to run, then `OS_ISRExit()` (8) takes longer to execute because a context switch is now needed (9), and an exception return instruction is executed (10).

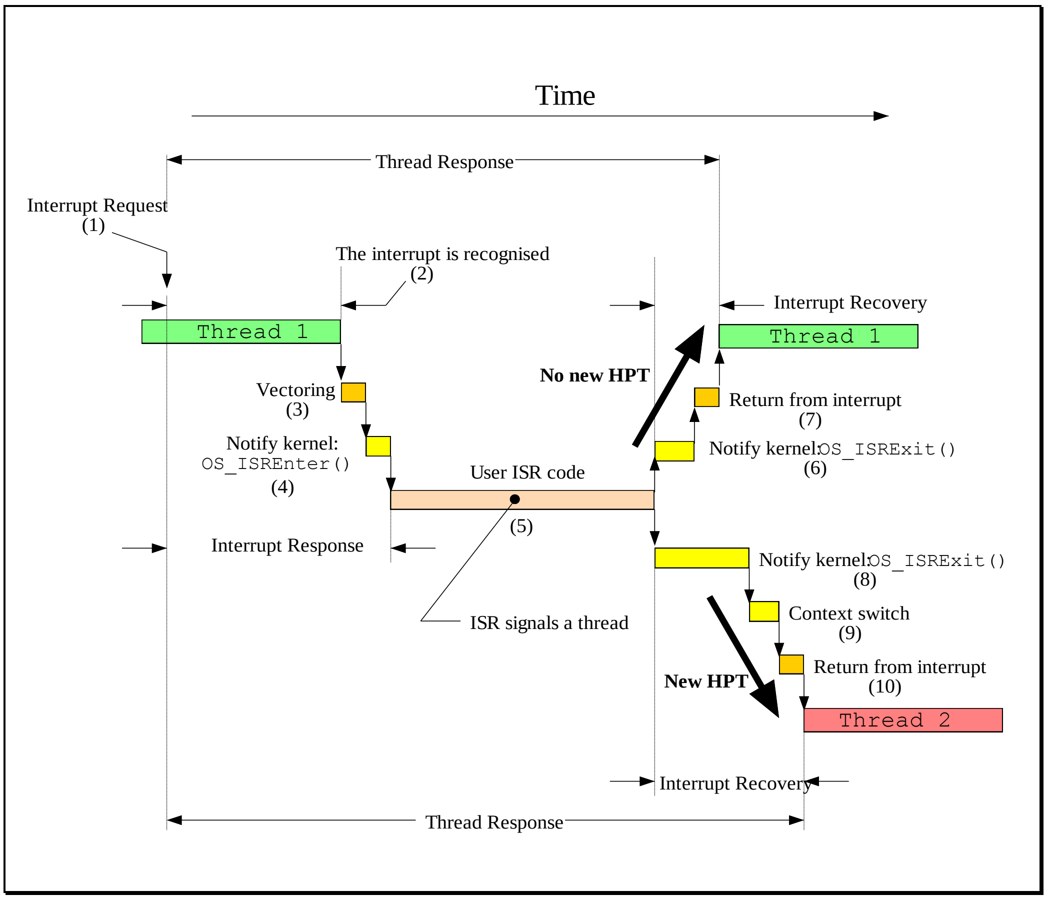

Figure R.2 – Interrupt operation of the RTOS

R.3.1 Clock Tick

The RTOS provides a periodic time source to keep track of time delays and timeouts. The faster the tick rate, the higher the overhead imposed on the system. The actual frequency of the clock tick depends on the desired tick resolution of your application.

The RTOS uses the Cortex®-M4's internal SysTick timer to generate ticks, so it cannot be used by the user. For this implementation, the clock tick period has been fixed at:

$$\boxed{\text{Clock Tick Period} = \mathbf{10\ ms}} \tag{R.1}$$

R.4 Interthread Communication & Synchronization

An RTOS normally provides many mechanisms to protect shared data and provide interthread communication. One simple way is to disable and enable interrupts through the macros `OS_EnterCritical()` and `OS_ExitCritical()`, respectively. You use these macros when two threads or a thread and an ISR need to share data.

This section discusses semaphores, which are normally used for synchronization and coordination. Operating systems also normally provide services to exchange information, such as message mailboxes and message queues.

R.4.1 Semaphores

A semaphore needs to be created before it can be used. Create a semaphore by calling `OS_SemaphoreCreate()` and specifying the initial count of the semaphore. The initial value of a semaphore can be between 0 and 4294967295. If you use the semaphore to signal the occurrence of one or more events, initialize the semaphore to 0. If you use the semaphore to access a shared resource, initialize the semaphore to 1 (i.e., use it as a binary semaphore). Finally, if the semaphore allows your application to obtain any one of *n* identical resources, initialize the semaphore to *n* and use it as a counting semaphore.

The RTOS provides three services to access semaphores: `OS_SemaphoreCreate()`, `OS_SemaphoreSignal()` and `OS_SemaphoreWait()`. Figure R.3 shows a flow diagram to illustrate the relationship between threads, ISRs, and a semaphore.

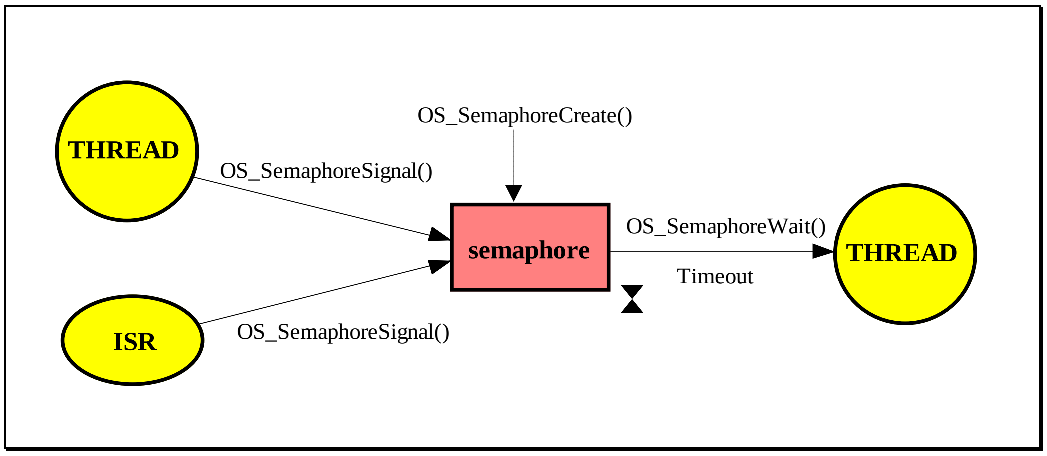

Figure R.3 – Relationships between threads, ISRs and a semaphore

As you can see from Figure R.3, a thread or an ISR can call `OS_SemaphoreSignal()`. However, only threads are allowed to call `OS_SemaphoreWait()`.

R.4.2 Creating a Semaphore, `OS_SemaphoreCreate()`

`OS_SemaphoreCreate()` returns a pointer that must be used in subsequent calls to manipulate the semaphore. The pointer is basically used as the semaphore's handle. If the RTOS cannot create the semaphore (because it has run out of internal resources), `OS_SemaphoreCreate()` returns a `NULL` pointer.

You should note that once a semaphore has been created, it cannot be deleted. It would be dangerous to delete a semaphore object if threads were waiting on the semaphore and / or relying on the presence of the semaphore.

R.4.3 Signalling a Semaphore, `OS_SemaphoreSignal()`

`OS_SemaphoreSignal()` checks to see if any threads are waiting on the semaphore. The highest priority thread waiting for the semaphore is made ready to run. The RTOS then checks to see if the thread made ready is now the highest priority thread ready to run. If it is, a context switch results (only if `OS_SemaphoreSignal()` is called from a thread) and the readied thread is executed. If the readied thread is not the highest priority thread, the thread that called `OS_SemaphoreSignal()` continues execution. If there are no threads waiting on the semaphore, the semaphore count simply gets incremented.

Note that a context switch does not occur if `OS_SemaphoreSignal()` is called by an ISR because context switching from an ISR can only occur when `OS_ISRExit()` is called at the completion of the ISR from the last nested ISR.

R.4.4 Waiting on a Semaphore, `OS_SemaphoreWait()`

`OS_SemaphoreWait()` starts by checking if the semaphore is available (its count is nonzero), in which case the count is decremented and the function returns to its caller with an error code indicating success. Obviously, if you want the semaphore, this is the outcome you are looking for.

If the semaphore count is zero, the calling thread needs to be put to sleep until another thread (or an ISR) signals the semaphore. `OS_SemaphoreWait()` allows you to specify a timeout value as one of its arguments (i.e., `timeout`). This feature is useful to avoid waiting indefinitely for the semaphore. If the value passed is nonzero, `OS_SemaphoreWait()` suspends the thread until the semaphore is signalled or the specified timeout period expires. Note that a timeout value of 0 indicates that the thread is willing to wait forever for the semaphore to be signalled.

Because the calling thread is no longer ready to run, the RTOS's scheduler is called to run the next highest priority thread that is ready to run. When the semaphore is signalled (or the timeout period expires) and the thread that called

`OS_SemaphoreWait()` is again the highest priority thread, the RTOS then checks to see if the thread is waiting for the semaphore. If the thread is still waiting for the semaphore, it must not have been signalled by an `OS_SemaphoreSignal()` call. Instead, the timeout period must have expired. In this case, the thread is removed from the wait list for the semaphore and an error code is returned to the thread that called `OS_SemaphoreWait()` to indicate that a timeout occurred. If the semaphore was signalled, then the thread that called `OS_SemaphoreWait()` can now conclude that it has the semaphore.

R.5 Initialization and Configuration

The RTOS needs a few constants declared in its `OS.h` file:

`OS_IDLE_THREAD_STACK_SIZE` declares the number of bytes used for the idle thread stack. Sufficient stack space must be allocated to accommodate for maximum interrupt nesting.

`OS_MAX_USER_THREADS` defines the number of user threads that you wish the RTOS to manage.

`OS_MAX_EVENTS` defines the maximum number of “event control blocks” that your application will create. Each semaphore requires an “event control block”.

R.5.1 RTOS Initialization

A requirement of the RTOS is that you call `OS_Init()` before you call any of its other services. `OS_Init()` initializes all the RTOS variables and data structures.

`OS_Init()` creates the idle thread `Idle()`, which is always ready to run. The priority of `Idle()` is always set to `OS_LOWEST_PRIORITY`.

R.5.2 Starting the RTOS

You start multithreading by calling `OS_Start()`. However, before you start the RTOS, you must create at least one of your application threads as shown in Listing R.5.

```
void main(void)
{
  // Initialize the RTOS
  OS_Init(CPU_CORE_CLK_HZ, true);
  .
  .
  // Create at least 1 thread using OS_ThreadCreate();
  .
  .
  // Start multithreading!
  OS_Start();
  // OS_Start() will not return
}
```

Listing R.5 – Initializing and starting the RTOS

When called, `OS_Start()` finds the `TCB` (from the ready list) of the highest priority thread that you have created. Then, `OS_Start()` executes a context switch, which forces the CPU to execute your thread's code.

R.6 RTOS Reference for Applications

This section provides a reference guide to the RTOS services that are provided to applications. Each of the user-accessible kernel services is presented in alphabetical order and the following information is provided for each of the services.

- The function prototype
- From where it is called
- A brief description
- A description of the arguments passed to the function
- A description of the return value(s)
- Specific notes and warnings on using the service
- An example of how to use the function

`OS_Init()`

Prototype:

```
void OS_Init(const uint32_t cpuCoreClk, const bool toggleLED);
```

Called from:

Startup code only

Description:

`OS_Init()` initializes the RTOS and must be called prior to calling `OS_Start()`, which actually starts multithreading.

Arguments:

`cpuCoreClk` is the value of the CPU core clock. This is used to set up the OS clock tick interrupt.

`toggleLED` will let the OS toggle the orange LED every second if true.

Return Value:

None

Notes / Warnings:

`OS_Init()` must be called before `OS_Start()`.

Example:

```
void main (void)
{
  // User code
  .
  // Initialize the RTOS
  OS_Init(CPU_CORE_CLK_HZ, true);
  // User code
  .
  // Start multithreading
  OS_Start();
}
```

OS_ISREnter()

Prototype:

`void OS_ISREnter(void);`

Called from:

ISR only

Description:

`OS_ISREnter()` notifies the RTOS that an ISR is being processed. This allows the RTOS to keep track of interrupt nesting. `OS_ISREnter()` is used in conjunction with `OS_ISRExit()`.

Arguments:

None

Return Value:

None

Notes / Warnings:

This function must not be called by thread-level code. The interrupt flag should be cleared immediately after calling this function as a matter of style.

Example:

```
void interrupt MyISR(void)
{
  // Notify RTOS of start of ISR
  OS_ISREnter();
  // Clear interrupt flag
  .
  .
  .
  OS_ISRExit();
}
```

OS_ISRExit()

Prototype:

void OS_ISRExit(**void**);

Called from:

ISR only

Description:

OS_ISRExit() notifies the RTOS that an ISR has completed. This allows the RTOS to keep track of interrupt nesting. OS_ISRExit() is used in conjunction with OS_ISREnter(). When the last nested interrupt completes, the RTOS calls the scheduler to determine if a higher priority thread has been made ready to run, in which case, the interrupt returns to the higher priority thread instead of the interrupted thread.

Arguments:

None

Return Value:

None

Notes / Warnings:

This function must not be called by thread-level code.

Example:

```
void interrupt MyISR(void)
{
  // Notify RTOS of start of ISR
  OS_ISREnter();
  // Clear interrupt flag
  .
  .
  .
  OS_ISRExit();
}
```

OS_SemaphoreCreate()

Prototype:

```
OS_ECB* OS_SemaphoreCreate(const uint32_t value);
```

Called from:

Thread or startup code

Description:

`OS_SemaphoreCreate()` creates and initializes a semaphore. A semaphore:

- allows a thread to synchronize with either an ISR or a thread,
- gains exclusive access to a resource, and
- signals the occurrence of an event.

Arguments:

`value` is the initial value of the semaphore and can be between `0` and `4294967295`.

Return Value:

A pointer to the event control block allocated to the semaphore. If no event control block is available, a `NULL` pointer is returned.

Notes / Warnings:

Semaphores must be created before they are used.

Example:

```
OS_ECB* UARTInUse

void main(void)
{
  .           // User code
  OS_Init(CPU_CORE_CLK_HZ, true);   // Initialize RTOS
  .
  UARTInUse = OS_SemaphoreCreate(1);// Create UART semaphore
  OS_Start();                       // Start multithreading
}
```

OS_SemaphoreSignal()

Prototype:

```
OS_ERROR OS_SemaphoreSignal(OS_ECB* const pEvent);
```

Called from:

Thread or ISR

Description:

A semaphore is signalled by calling `OS_SemaphoreSignal()`. If the semaphore value is `1` or more, it is incremented and `OS_SemaphoreSignal()` returns to its caller. If threads are waiting for the semaphore to be signalled, `OS_SemaphoreSignal()` removes the highest priority thread pending for the semaphore from the waiting list and makes this thread ready to run. The scheduler is then called to determine if the awakened thread is now the highest priority thread ready to run.

Arguments:

`pEvent` is a pointer to the semaphore. This pointer is returned to your application when the semaphore is created [see `OS_SemaphoreCreate()`].

Return Value:

`OS_SemaphoreSignal()` returns one of two error codes:

- `OS_NO_ERROR` if the semaphore was signalled successfully.
- `OS_SEMAPHORE_OVERFLOW` if the semaphore count overflowed.

Notes / Warnings:

Semaphores must be created before they are used.

Example:

```
OS_ECB* UARTInUse

void UARTThread(void* pData)
{
  OS_ERROR error;

  for (;;)
  {
    // User code
    .
    .
    error = OS_SemaphoreSignal(UARTInUse);
    if (error == OS_NO_ERROR)
    {
      // Semaphore signalled
      .
      .
    }
    else
    {
      // Semaphore has overflowed
      .
      .
    }
  }
}
```

OS_SemaphoreWait()

Prototype:

```
OS_ERROR OS_SemaphoreWait(OS_ECB* const pEvent, const uint32_t timeout);
```

Called from:

Thread only

Description:

`OS_SemaphoreWait()` is used when a thread wants exclusive access to a resource, needs to synchronize its activities with an ISR or a thread, or is waiting until an event occurs. If a thread calls `OS_SemaphoreWait()` and the value of the semaphore is greater than `0`, `OS_SemaphoreWait()` decrements the semaphore and returns to its caller. However, if the value of the semaphore is equal to `0`, `OS_SemaphoreWait()` places the calling thread in the waiting list for the semaphore. The thread will thus wait until a thread or an ISR signals the semaphore or the specified timeout expires. If the semaphore is signalled before the timeout expires, the RTOS resumes the highest priority thread waiting for the semaphore.

Arguments:

`pEvent` is a pointer to the semaphore. This pointer is returned to your application when the semaphore is created [see `OS_SemaphoreCreate()`].

`timeout` allows the thread to resume execution if the semaphore is not acquired within the specified number of clock ticks. A `timeout` value of `0` indicates that the thread will wait forever for the message. The maximum timeout is `4294967295` clock ticks. The timeout value is not synchronized with the clock tick. The timeout count begins decrementing on the next clock tick, which could potentially occur immediately.

Return Value:

`OS_SemaphoreWait()` returns one of two error codes:

- `OS_NO_ERROR` if the semaphore was available.
- `OS_TIMEOUT` if the semaphore was not signalled within the specified timeout.

Notes / Warnings:

Semaphores must be created before they are used. This function cannot be called from an ISR.

Example:

```
OS_ECB* UARTInUse

void UARTThread(void* pData)
{
  OS_ERROR error;

  for (;;)
  {
    // User code
    .
    .
    error = OS_SemaphoreWait(UARTInUse, 0);
    // User code
    .
    .
  }
}
```

OS_Start()

Prototype:

`void OS_Start(void);`

Called from:

Startup code only

Description:

`OS_Start()` starts the RTOS multithreading.

Arguments:

None

Return Value:

None

Notes / Warnings:

`OS_Init()` must be called prior to calling `OS_Start()`. `OS_Start()` should only be called once by your application code. If you do call `OS_Start()` more than once, it will not do anything on the second and subsequent calls. `OS_Start()` will never return to its caller.

Example:

```
void main (void)
{
  // User code
  .
  // Initialize the RTOS
  OS_Init(CPU_CORE_CLK_HZ, true);
  // User code
  .
  // Start multithreading
  OS_Start();
}
```

OS_ThreadCreate()

Prototype:

```
OS_ERROR OS_ThreadCreate(void (*thread)(void* pd),
  void* pData, void* pStack, const uint8_t priority);
```

Called from:

Thread or startup code

Description:

`OS_ThreadCreate()` creates a thread so it can be managed by the RTOS. Threads can be created either prior to the start of multithreading or by a running thread. A thread cannot be created by an ISR. A thread must be written as an infinite loop, as shown below, and must not return.

Arguments:

`thread` is a pointer to the thread's code.

`pData` is a pointer to an optional data area used to pass parameters to the thread when it is created. Where the thread is concerned, it thinks it was invoked and passed the argument `pData` as follows:

```
void Thread(void* pData)
{
  .      // Do something with 'pData'
  for (;;)
  {
    // Thread body, always an infinite loop.
    .
    .
    // Must call one of the following services:
    // OS_SemaphoreWait()
    // OS_TimeDelay()
    // OS_ThreadDelete()   // (Delete self)
    .
    .
  }
}
```

`pStack` is a pointer to the thread's top-of-stack. The stack is used to store local variables, function parameters, return addresses, and CPU registers during an interrupt. The size of the stack is determined by the thread's requirements and the anticipated interrupt nesting. Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the thread itself and all nested functions, as well as requirements for interrupts (accounting for nesting). `pStack` thus needs to point to the highest valid memory location on the stack.

`priority` is the thread priority. A unique priority number must be assigned to each thread and the lower the number, the higher the priority.

Return Value:

`OS_ThreadCreate()` returns one of the following error codes:

- `OS_NO_ERROR` if the function was successful.
- `OS_PRIORITY_EXISTS` if the requested priority already exists.
- `OS_PRIORITY_INVALID` if `priority` is higher than `OS_LOWEST_PRIORITY`.
- `OS_NO_MORE_TCBS` if the RTOS doesn't have any more `TCB`s to assign.

Notes / Warnings:

A thread cannot be created by an ISR.

A thread must always invoke one of the services provided by the RTOS to either wait for time to expire, or wait for an event to occur (wait on a semaphore). This allows other threads to gain control of the CPU.

You should not use thread priority `OS_LOWEST_PRIORITY` because it is reserved for use by the RTOS for the idle thread. This leaves you with up to 31 application threads.

Example:

You can create a generic thread that can be instantiated more than once. For example, a thread that handles a serial port could be passed the address of a data structure that characterizes the specific port (i.e., port address, baud rate).

```
uint32_t UART2Stack[1024];

// Data structure containing COM port
//   specific data for UART2
UART_DATA UART2Data;

uint32_t UART3Stack[1024];

// Data structure containing COM port
//   specific data for UART3
UART_DATA UART3Data;

void main(void)
{
  OS_ERROR error;
  .
  // Initialize the RTOS
  OS_Init(CPU_CORE_CLK_HZ, true);
  .
  error = OS_ThreadCreate(UARTThread,
                          (void*)&UART2Data,
                          &UART2Stack[1023],
                          10);
  error = OS_ThreadCreate(UARTThread,
                          (void*)&UART3Data,
                          &UART3Stack[1023],
                          11);
  .
  // Start multithreading
  OS_Start();
}

// Generic UART thread
void UARTThread(void* pData)
{
  for (;;)
  {
    .  // Thread code
    .
  }
}
```

OS_ThreadDelete()

Prototype:

```
OS_ERROR OS_ThreadDelete(uint8_t priority);
```

Called from:

Thread only

Description:

`OS_ThreadDelete()` deletes a thread by specifying the priority number of the thread to delete. The calling thread can be deleted by specifying its own priority number or `OS_PRIORITY_SELF` (if the thread doesn't know its own priority number). The deleted thread is returned to the dormant state. The deleted thread can be created by calling `OS_ThreadCreate()` to make the thread active again.

Arguments:

`priority` is the priority number of the thread to delete. You can delete the calling thread by passing `OS_PRIORITY_SELF`, in which case, the next highest priority thread is executed.

Return Value:

`OS_ThreadDelete()` returns one of the following error codes:

- `OS_NO_ERROR` if the thread was deleted.
- `OS_THREAD_DELETE_ERROR` if the thread to delete does not exist.
- `OS_THREAD_DELETE_IDLE` if you tried to delete the idle thread.
- `OS_PRIORITY_INVALID` if you specified a thread priority higher than `OS_LOWEST_PRIORITY`.
- `OS_THREAD_DELETE_ISR` if you tried to delete a thread from an ISR.

Notes / Warnings:

`OS_ThreadDelete()` verifies that you are not attempting to delete the RTOS idle thread.

You must be careful when you delete a thread that owns resources.

Example:

```
void ThreadX(void* pData)
{
  OS_ERROR error;

  for (;;)
  {
    .
    .
    // Delete thread with priority 10
    error = OS_ThreadDelete(10);
    if (error == OS_NO_ERR)
    {
      // Thread was deleted
      .
    }
    .
    .
  }
}
```

OS_TimeDelay()

Prototype:

```
void OS_TimeDelay(const uint32_t ticks);
```

Called from:

Thread only

Description:

`OS_TimeDelay()` allows a thread to delay itself for a number of clock ticks. Rescheduling always occurs when the number of clock ticks is greater than zero. Valid delays range from `0` to `4294967295` ticks. A delay of `0` means that the thread is not delayed and `OS_TimeDelay()` returns immediately to the caller. The actual delay time depends on the tick rate.

Arguments:

`ticks` is the number of clock ticks to delay the current thread.

Return Value:

None

Notes / Warnings:

To ensure that a thread delays for the specified number of ticks, you should consider using a delay value that is one tick higher. For example, to delay a thread for at least 10 ticks, you should specify a value of 11.

Example:

```
void ThreadX(void* pData)
{
  for (;;)
  {
    .
    // Delay thread for at least 10 clock ticks
    OS_TimeDelay(11);
    .
  }
}
```

OS_TimeGet()

Prototype:

`uint32_t OS_TimeGet(void);`

Called from:

Thread or ISR

Description:

`OS_TimeGet()` obtains the current value of the system clock. The system clock is a 32-bit counter that counts the number of clock ticks since power was applied or since the system clock was last set.

Arguments:

None

Return Value:

The current system clock value (in number of ticks).

Notes / Warnings:

None

Example:

```
void ThreadX(void* pData)
{
  uint32_t clock;

  for (;;)
  {
    .
    // Get current value of system clock
    clock = OS_TimeGet();
    .
  }
}
```

OS_TimeSet()

Prototype:

```
void OS_TimeSet(const uint32_t ticks);
```

Called from:

Thread or ISR

Description:

`OS_TimeSet()` sets the system clock. The system clock is a 32-bit counter that counts the number of clock ticks since power was applied or since the system clock was last set.

Arguments:

`ticks` is the desired value for the system clock, in ticks.

Return Value:

None

Notes / Warnings:

None

Example:

```
void ThreadX(void* pData)
{
  for (;;)
  {
    .
    // Reset the system clock
    OS_TimeSet(0);
    .
  }
}
```

`OS_DisableInterrupts()` and `OS_EnableInterrupts()`

Prototype:

Macros

Called from:

Thread or ISR

Description:

`OS_DisableInterrupts()` and `OS_EnableInterrupts()` are macros used to disable and enable, respectively, the processor's interrupts.

Arguments:

None

Return Value:

None

Notes / Warnings:

These macros must be used in pairs.

Example:

```
void ThreadX(void* pData)
{
  for (;;)
  {
    .
    OS_DisableInterrupts();   // Disable interrupts
    .
    .                         // Access critical code
    .
    OS_EnableInterrupts();    // Enable interrupts
    .
  }
}
```