Messaging and Synchronizing in RTX

Ravi Suppiah
Lecturer, NUS SoC

Inter-Task Communication

- Inter-task communication primitives are of crucial importance to OS design
- Several mechanisms or primitives are provided by RTX
  - Flags
  - Events
  - Mutex
  - Semaphore
  - Mailbox
- The first four provide **synchronization**
- The last one provides **data communication**
- We will have a close look at each one of them one by one.

Flags

- Thread Flags are not created.
- A 32-bit word with 31 Thread Flags already exist automatically within each thread.

0	TF30	TF29	TF29		TF2	TF1	TF0

- One thread sets TFs in another thread (addressed by its thread ID).
- A thread ID variable must first be declared as a global.

```
osThreadId_t redLED_Id;
```

- The thread ID is returned when we call osThreadNew().

```
redLED_Id = osThreadNew(led_red_thread, NULL, NULL);
```

Flags

```
uint32_t osThreadFlagsSet ( osThreadId_t thread_id,
                            uint32_t     flags
                          )
```

Parameters

[in]	**thread_id**	thread ID obtained by **osThreadNew** or **osThreadGetId**.
[in]	**flags**	specifies the flags of the thread that shall be set.

Returns

thread flags after setting or error code if highest bit set.

The function **osThreadFlagsSet** sets the thread flags for a thread specified by parameter *thread_id*. It returns the flags set, or an error code if highest bit is set (refer to **Flags Functions Error Codes**). This function may be used also within interrupt service routines. Threads waiting for a flag to be set will resume from **BLOCKED** state.

```
uint32_t osThreadFlagsWait ( uint32_t flags,
                             uint32_t options,
                             uint32_t timeout
                           )
```

Parameters

[in]	**flags**	specifies the flags to wait for.
[in]	**options**	specifies flags options (osFlagsXxxx).
[in]	**timeout**	**Timeout Value** or 0 in case of no time-out.

Returns

thread flags before clearing or error code if highest bit set.

The function **osThreadFlagsWait** suspends the execution of the currently **RUNNING** thread until any or all of the thread flags specified with the parameter *flags* are set. When these thread flags are already set, the function returns instantly. Otherwise the thread is put into the state **BLOCKED**.

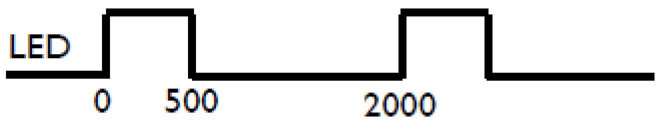
The parameter *options* specifies the wait condition:

Option	
osFlagsWaitAny	Wait for any flag (default).
osFlagsWaitAll	Wait for all flags.
osFlagsNoClear	Do not clear flags which have been specified to wait for.

Flags

```
//Thread 1
void ledOn(void constant *argument) {
    for (;;) {
        LED_On();
        osThreadFlagsSet(tid_ledOff, 0x0001); //signal ledOffthread
        osDelay(2000);
    }
}

// Thread 2
void ledOff(void constant *argument) {
    for (;;) {
        // wait for signal from ledOnthread
        osThreadFlagsWait(0x0001, osFlagsWaitAny, osWaitForever);
        osDelay(500);
        LED_Off();
    }
}
```



Flags - Thread3 must wait for signals from both Thread1 and Thread2

```
osThreadId_t tid1;//three threads
osThreadId_t tid2;
osThreadId_t tid3;

void thread1 (void *argument)
{
    while (1) {
    osThreadFlagsSet(tid3, 0x0001); /* signal thread 3 */
    ….
    }
}
void thread2 (void *argument)
{
    while (1) {
    osThreadFlagsSet(tid3, 0x0002); /* signal thread 3 */
    ….
    }
}
void thread3 (void *argument)
{
    uint32_t flags;
    while (1) {
    //wait for signals from boththread1 and thread2
    flags = osThreadFlagsWait(0x0003, osFlagsWaitAll, osWaitForever);
…   //continue processing
    }
}
```

NUS National University of Singapore

ARM

Events

- Similar to Flags except that they are not tied to any particular Task.
- Need to explicitly create a new Events Flag object.
- Each object will have 31 Event Flags.

```
osEventFlagsId_t osEventFlagsNew ( const osEventFlagsAttr_t * attr )
```

Parameters

[in] **attr** event flags attributes; NULL: default values.

Returns

event flags ID for reference by other functions or NULL in case of error.

The function **osEventFlagsNew** creates a new event flags object that is used to send events across threads and returns the pointer to the event flags object identifier or *NULL* in case of an error. It can be safely called before the RTOS is started (call to **osKernelStart**), but not before it is initialized (call to **osKernelInitialize**).

Events

```
uint32_t osEventFlagsSet ( osEventFlagsId_t  ef_id,
                           uint32_t          flags
                         )
```

Parameters

[in] **ef_id** event flags ID obtained by **osEventFlagsNew**.

[in] **flags** specifies the flags that shall be set.

Returns

event flags after setting or error code if highest bit set.

The function **osEventFlagsSet** sets the event flags specified by the parameter *flags* in an event flags object specified by parameter *ef_id*. All threads waiting for the flag set will be notified to resume from **BLOCKED** state. The function returns the event flags after setting or an error code (highest bit is set, refer to **Flags Functions Error Codes**).

```
uint32_t osEventFlagsWait ( osEventFlagsId_t  ef_id,
                            uint32_t          flags,
                            uint32_t          options,
                            uint32_t          timeout
                          )
```

Parameters

[in] **ef_id** event flags ID obtained by **osEventFlagsNew**.

[in] **flags** specifies the flags to wait for.

[in] **options** specifies flags options (osFlagsXxxx).

[in] **timeout** **Timeout Value** or 0 in case of no time-out.

Returns

event flags before clearing or error code if highest bit set.

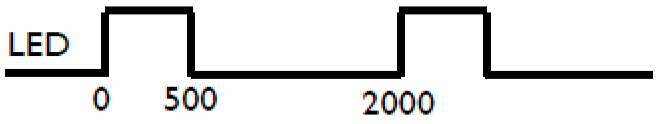
The function **osEventFlagsWait** suspends the execution of the currently **RUNNING** thread until any or all event flags specified by the parameter *flags* in the event object specified by parameter *ef_id* are set. When these event flags are already set, the function returns instantly. Otherwise, the thread is put into the state **BLOCKED**.

The *options* parameter specifies the wait condition:

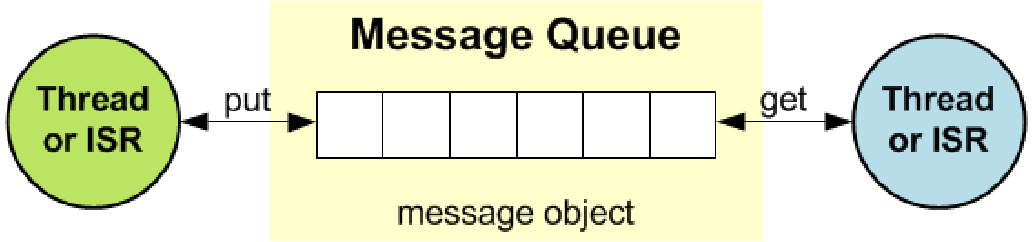
Option	
osFlagsWaitAny	Wait for any flag (default).
osFlagsWaitAll	Wait for all flags.
osFlagsNoClear	Do not clear flags which have been specified to wait for.

Events

```c
osEventFlagsId_t led_flag;
void main_app(void constant *argument)
{
    led_flag= osEventFlagsNew(NULL); //create the event flag
}
void ledOn(void constant *argument)
{
    for (;;)
    {
        LED_On();
        osEventFlagsSet(led_flag, 0x0001); //signal ledOffthread
        osDelay(2000);
    }
}
void ledOff(void constant *argument)
{
    for (;;)
    { // wait for signal from ledOnthread
        osEventFlagsWait(led_flag, 0x0001, osFlagsWaitAny, osWaitForever);
        osDelay(500);
        LED_Off();
    }
}
```



Message Queue



- **Message passing** is another basic communication model between threads.

- In the message passing model, one thread sends data explicitly, while another thread receives it.

- The operation is more like some kind of I/O rather than a direct access to information to be shared.

- Using message queue functions, you can control, send, receive, or wait for messages. The data to be passed can be of integer or pointer type:

Message Queue

```
osMessageQueueId_t osMessageQueueNew ( uint32_t                      msg_count,
                                       uint32_t                      msg_size,
                                       const osMessageQueueAttr_t *  attr
                                     )
```

Parameters

[in]	**msg_count**	maximum number of messages in queue.
[in]	**msg_size**	maximum message size in bytes.
[in]	**attr**	message queue attributes; NULL: default values.

Returns

message queue ID for reference by other functions or NULL in case of error.

The function **osMessageQueueNew** creates and initializes a message queue object. The function returns a message queue object identifier or *NULL* in case of an error.

The function can be called after kernel initialization with **osKernelInitialize**. It is possible to create message queue objects before the RTOS kernel is started with **osKernelStart**.

The total amount of memory required for the message queue data is at least `msg_count * msg_size`. The *msg_size* is rounded up to a double even number to ensure 32-bit alignment of the memory blocks.

Message Queue

```
osStatus_t osMessageQueuePut ( osMessageQueueId_t  mq_id,
                               const void *        msg_ptr,
                               uint8_t             msg_prio,
                               uint32_t            timeout
                             )
```

Parameters

[in]	**mq_id**	message queue ID obtained by **osMessageQueueNew**.
[in]	**msg_ptr**	pointer to buffer with message to put into a queue.
[in]	**msg_prio**	message priority.
[in]	**timeout**	**Timeout Value** or 0 in case of no time-out.

Returns

status code that indicates the execution status of the function.

The blocking function **osMessageQueuePut** puts the message pointed to by *msg_ptr* into the the message queue specified by parameter *mq_id*. The parameter *msg_prio* is used to sort message according their priority (higher numbers indicate a higher priority) on insertion.

The parameter *timeout* specifies how long the system waits to put the message into the queue. While the system waits, the thread that is calling this function is put into the **BLOCKED** state. The parameter **timeout** can have the following values:

Message Queue

```
osStatus_t osMessageQueueGet ( osMessageQueueId_t  mq_id,
                               void *              msg_ptr,
                               uint8_t *           msg_prio,
                               uint32_t            timeout
                             )
```

Parameters

[in]	**mq_id**	message queue ID obtained by **osMessageQueueNew**.
[out]	**msg_ptr**	pointer to buffer for message to get from a queue.
[out]	**msg_prio**	pointer to buffer for message priority or NULL.
[in]	**timeout**	**Timeout Value** or 0 in case of no time-out.

Returns

status code that indicates the execution status of the function.

The function **osMessageQueueGet** retrieves a message from the message queue specified by the parameter *mq_id* and saves it to the buffer pointed to by the parameter *msg_ptr*. The message priority is stored to parameter *msg_prio* if not token{NULL}.

The parameter *timeout* specifies how long the system waits to retrieve the message from the queue. While the system waits, the thread that is calling this function is put into the **BLOCKED** state. The parameter **timeout** can have the following values:

Message Queue

- Lets look an example on how we can use Message Queues for the Mini-Project.

- We first need to declare a Message Data Structure.

```
typedef struct
{
  uint8_t cmd;
  uint8_t data;
}myDataPkt;
```

- We then declare a new MessageQueue Id.

```
osMessageQueueId_t redMsg, greenMsg, blueMsg;
```

Message Queue

- We then create a new Message Queue for each led_thread in main().
- MSG_COUNT is defined as 1 in this example.

```
redMsg = osMessageQueueNew(MSG_COUNT, sizeof(myDataPkt), NULL);
greenMsg = osMessageQueueNew(MSG_COUNT, sizeof(myDataPkt), NULL);
blueMsg = osMessageQueueNew(MSG_COUNT, sizeof(myDataPkt), NULL);
```

- We are now ready to use the Message Queue.
- We will first modify the control_thread to send out messages targeting different LED's.
- The format of the cmd:data will be as such:

CMD	DATA
0x01: RED	0x01: Blink
0x01: GREEN	0x01: Blink
0x01: BLUE	0x01: Blink

Message Queue

- The control_thread will now send out messages to the other threads through their own message queues.

```
void control_thread (void *argument) {

  myDataPkt myData;

  myData.cmd = 0x01;
  myData.data = 0x01;
  // ...
  for (;;) {

    osMessageQueuePut(redMsg, &myData, NULL, 0);
    osDelay(2000);

    osMessageQueuePut(greenMsg, &myData, NULL, 0);
    osDelay(2000);

    osMessageQueuePut(blueMsg, &myData, NULL, 0);
    osDelay(2000);

    osMessageQueuePut(redMsg, &myData, NULL, 0);
    osMessageQueuePut(greenMsg, &myData, NULL, 0);
    osMessageQueuePut(blueMsg, &myData, NULL, 0);
    osDelay(2000);
  }
}
```

Message Queue

- Each of the threads will now receive a message in their respective message queues and process it accordingly.

```
void led_red_thread (void *argument) {

  myDataPkt myRxData;
  for (;;) {
    osMessageQueueGet(redMsg, &myRxData, NULL, osWaitForever);
    if(myRxData.cmd == 0x01 && myRxData.data == 0x01)
    {
      ledControl(RED_LED, led_on);
      osDelay(1000);
      ledControl(RED_LED, led_off);
      osDelay(1000);
    }
  }
}
```

```
void led_blue_thread (void *argument) {

  myDataPkt myRxData;
  for (;;) {
    osMessageQueueGet(blueMsg, &myRxData, NULL, osWaitForever);
    if(myRxData.cmd == 0x01 && myRxData.data == 0x01)
    {
      ledControl(BLUE_LED, led_on);
      osDelay(1000);
      ledControl(BLUE_LED, led_off);
      osDelay(1000);
    }
  }
}
```

```
void led_green_thread (void *argument) {

  myDataPkt myRxData;
  for (;;) {
    osMessageQueueGet(greenMsg, &myRxData, NULL, osWaitForever);
    if(myRxData.cmd == 0x01 && myRxData.data == 0x01)
    {
      ledControl(GREEN_LED, led_on);
      osDelay(1000);
      ledControl(GREEN_LED, led_off);
      osDelay(1000);
    }
  }
}
```

Message Queue

- The final observation will be as such:
  - RED -> GREEN -> BLUE -> WHITE -> RED -> GREEN ->…

- Though it may seem that using Thread Flags or Event Flags would be easier, the biggest advantage of using Message Queues is that the data packet can contain a lot more information than what is shown in this example.

- As such, it is possible to use Message Queues to allow threads to communicate and synchronize with each other while transferring large amounts of data at the same time.

Sharing Data Safely

- Preemption and interrupt could contaminate data:
  - One writer and at least one reader scenario
    - Data overwritten partway through being read - Writer and reader might interrupt each other
  - More than one writer, at least one reader scenario
    - Data overwritten partway through being read - Writer and reader might interrupt each other
    - Data overwritten partway through being written - Writers interrupt each other
- Is the read/write operation indivisible/atomic?
  - Yes, then problem solved – but what if not?
- Race condition
  - Anomalous behaviour due to unexpected critical dependence on the relative timing of events
  - Depends on the *relative timing* of the read and write operations
- Critical section
  - A section of code which creates a possible race condition
    - Any access to a shared data structure is a critical section of code
  - Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use

Atomicity, ARM ISA and Shared Memory

- ARM is a Load/Store architecture – variables can only be modified in registers, not in memory
- Any memory-resident variable must be accessed with at least three instructions: load, modify, store.
  - This creates a **critical section** from the load instruction to the store instruction (inclusive)
  - It's not just multi-element data structures – **ANYTHING** in memory (even bytes or words) is vulnerable
- Tasks communicating with shared memory are vulnerable to **race conditions**
- Any variables used in shared memory communication must be protected

General Solutions Based on RTX

- Use the primitives introduced to avoid data race

- Single Writer
- Buffer data so reader and writer don't access the same copy
  - Do-it-yourself buffering
  - Message Queues.

- Multiple writers
- Make sure only one writer accesses the shared variable
  - Use Mutex
  - Or disable preemption/interrupt (Disable Cortex-M interrupt or lock/unlock RTX)
- Also double check your tasks, better to make them re-entrant unless you have good reasons not to

Reentrant Function

- In multithreaded programs, functions called from more that one thread must use separate thread-specific instances of any data that should not be shared with other threads.

- This happens automatically when only automatic or dynamic allocation is used within the function.

- This creates a class of reentrant functions that inherently thread-safe because ever entry to the function creates new instances of its objects.

- Static objects, however, can never be thread-specific, since they have only a single instance.

The End

- You will explore Thread Flags, Event Flags and Mailboxes in the Lab. ☺