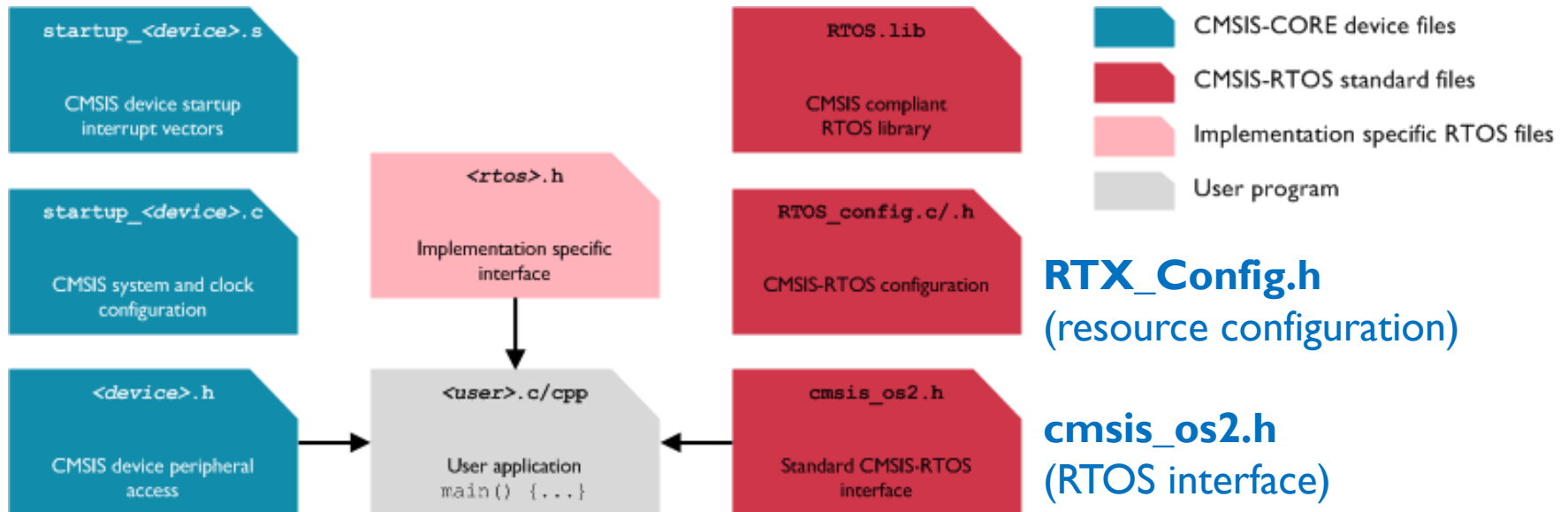
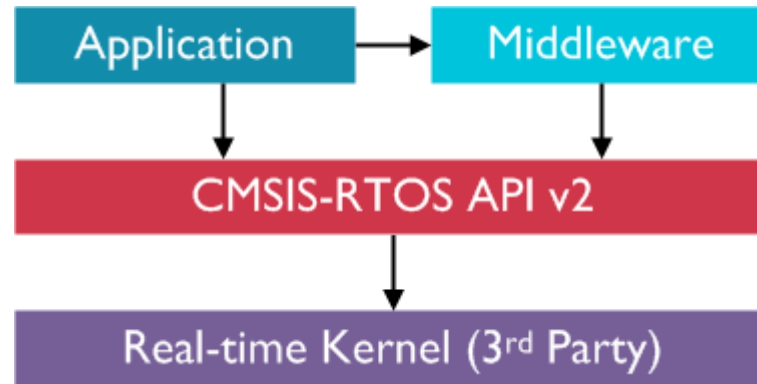


CMSIS Real Time Operating System (Based on Free RTOS)

References: <HTTPS://developer.mbed.org/handbook/CMSIS-RTOS>
<http://www.keil.com/pack/doc/CMSIS/RTOS2/html/index.html>

uVision5 Books Pane: “MDK-ARM Getting Started” (PDF), CMSIS-RTOS2 (pp26-36)
Keil directory: <C:/Keil/ARM/PACK/ARM/CMSIS/5.3.0/CMSIS/RTOS2>
(user code templates, examples, documentation)

CMSIS-RTOS2 Implementation (v2.x)



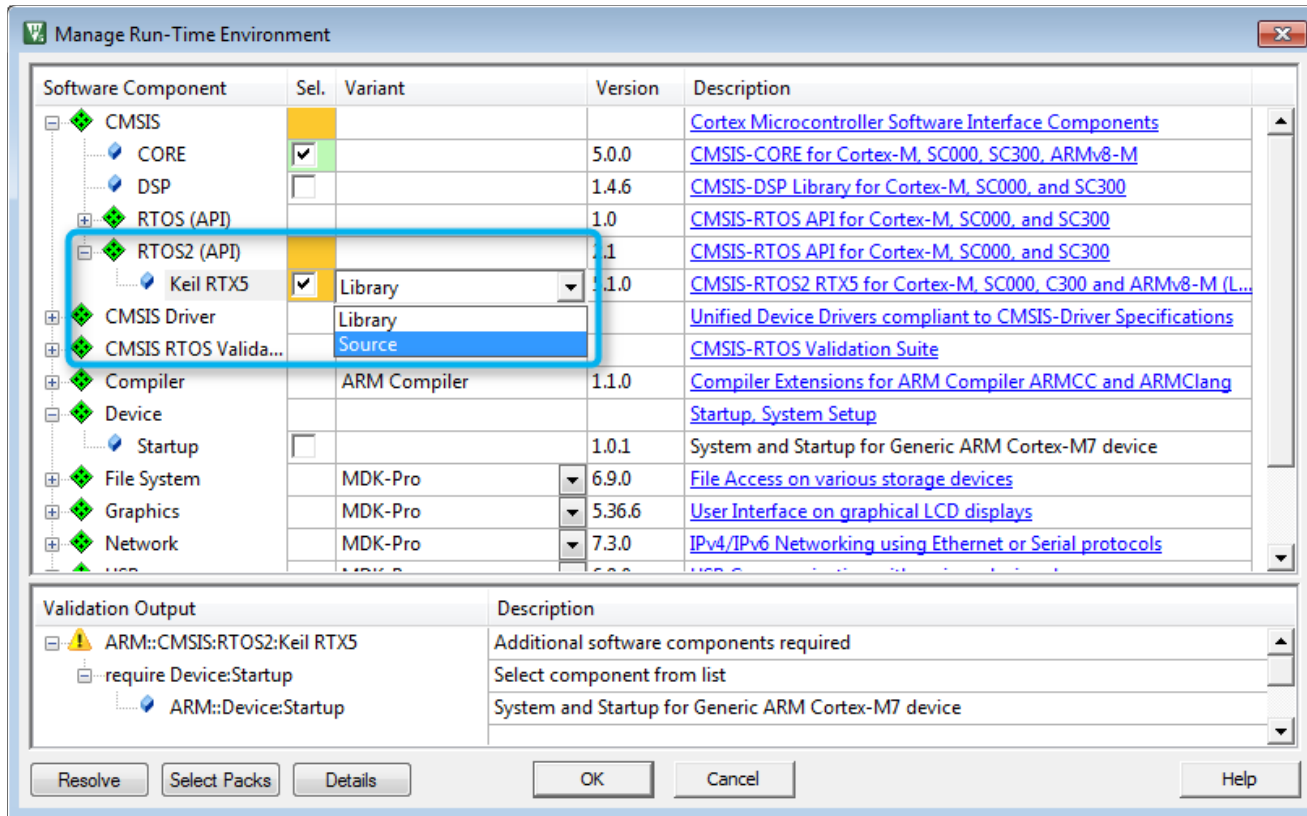
CMSIS-RTOS2 (*RTOS version 2*) API

- ▶ **Thread Management** allows you to define, create, and control threads.
- ▶ **Interrupt Service Routines** (ISR) can call some CMSIS-RTOS functions.
 - ▶ When a CMSIS-RTOS function cannot be called from an ISR context, it rejects the invocation and returns an error code.
- ▶ Three event types support communication between multiple threads and/or ISR:
 - ▶ **Thread Flags**: may be used to indicate specific conditions to a thread.
 - ▶ **Event Flags**: may be used to indicate events to a thread or ISR.
 - ▶ **Messages**: can be sent to a thread or an ISR. Messages are buffered in a queue.
- ▶ **Mutex Management** and **Semaphores** are incorporated.
- ▶ CPU time can be scheduled with the following functionalities:
 - ▶ A *timeout* parameter is incorporated in many CMSIS-RTOS functions to avoid system lockup. When a timeout is specified, the system waits until a resource is available or an event occurs. While waiting, other threads are scheduled.
 - ▶ The **osDelay** and **osDelayUntil** functions put a thread into the **WAITING** state for a specified period of time.
 - ▶ The **osThreadYield** provides co-operative thread switching and passes execution to another thread of the same priority.
- ▶ **Timer Management** functions are used to trigger the execution of functions.



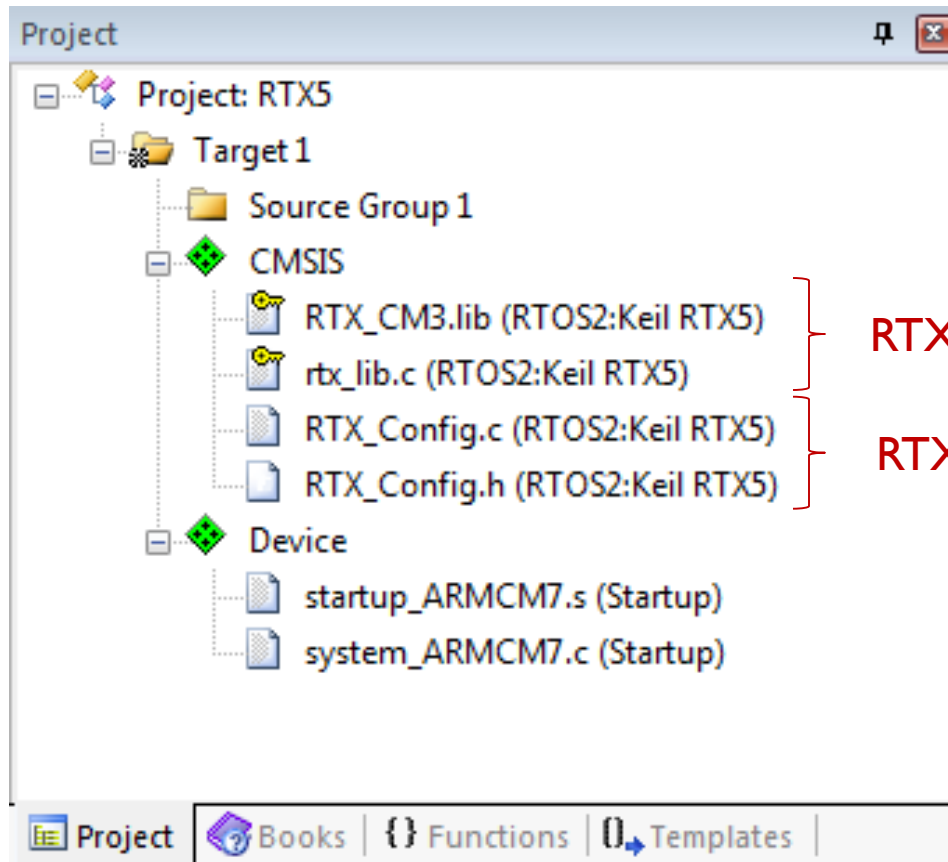
Using CMSIS-RTOS2 in a project

Create the project and add RTOS2 from *Manage Run-Time Environment* during project creation or from *Project > Manage > Run-Time Environment*



Select CMSIS-RTOS2
Library
or
Source (all .c files)

Created CMSIS-RTOS2 project



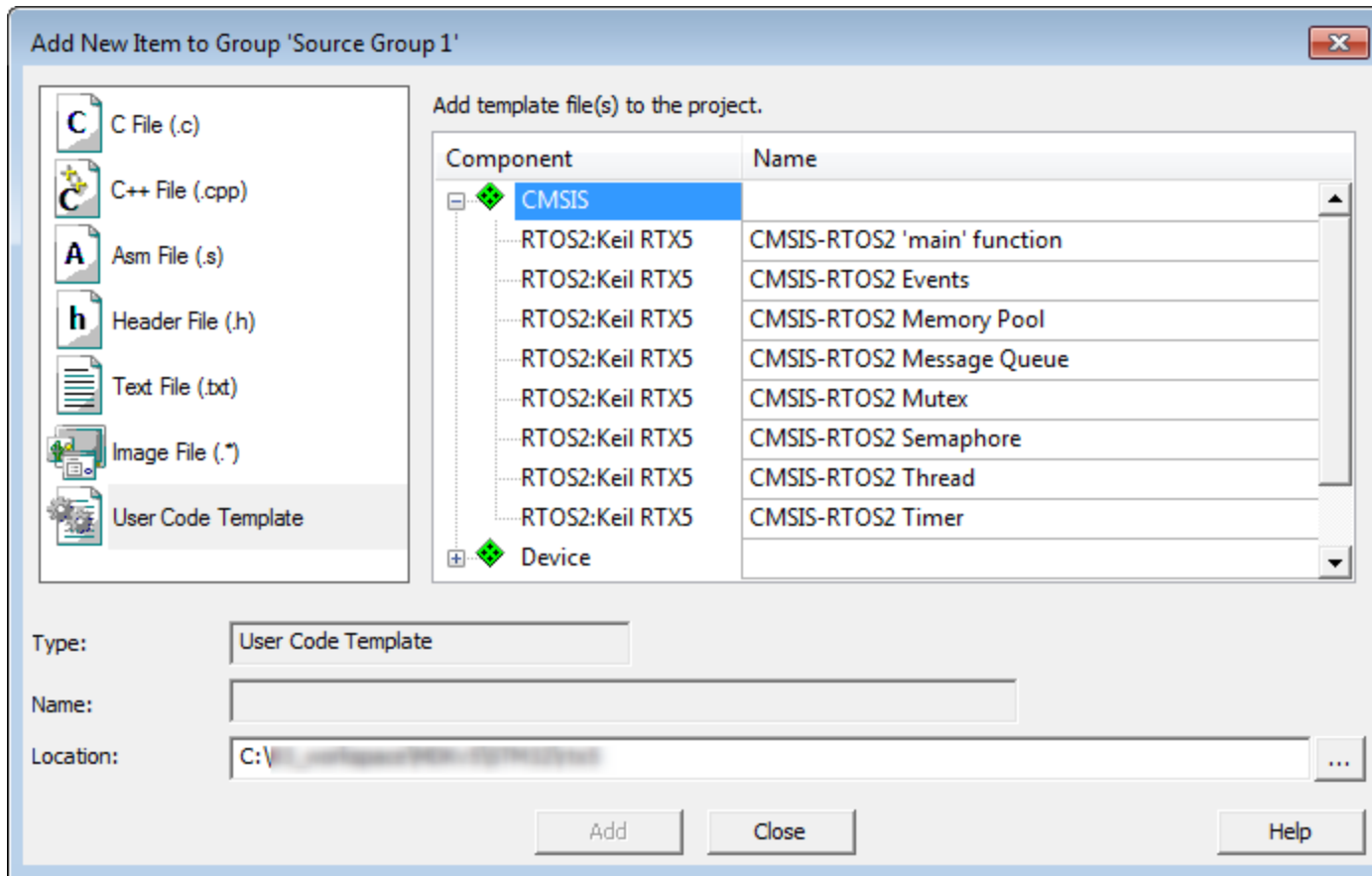
RTX function library

RTX configuration file

Edit RTX_Config.h to configure RTX kernel for this particular application.

#include "cmsis_os2.h" in source (RTOS2 API)

Templates can be used to create sources



Recommended order of program operations

In `main()`:

1. Initialize/configure hardware:
 - ▶ Peripherals, memory, pins, clocks, interrupts.
2. Configure system core clock
 - ▶ Optional: **SystemClock_Config()** for 80MHz clock
 - ▶ **SystemCoreClockUpdate()**; sets global variable `SystemCoreClock` used to configure SysTick timer.
3. Run **osKernelInitialize** to initialize CMSIS-RTOS kernel.
4. Run **osThreadNew** to create at least one thread *app_main*
 - ▶ RTOS scheduler will execute this thread when Kernel starts.
 - ▶ Use *app_main* to create “application” threads.
 - ▶ Alternatively, can create all threads in `main()`.
5. Run **osKernelStart** to start RTOS scheduler.
 - ▶ This function does not return in case of successful execution.
 - ▶ Any application code after **osKernelStart** will not be executed unless **osKernelStart** fails.

Starting the RTOS2 Kernel and Scheduler

```
#include "cmsis_os2.h"
osThreadId_t tid_phaseA;           // Thread id of thread "phaseA"

void phaseA (void *argument) {     // Some application thread
    ...some processing
}

void app_main (void *argument) {
    tid_phaseA = osThreadNew (phaseA, NULL, NULL); // Create thread "phaseA"
    osDelay(osWaitForever);                 // app_main never ready again
    while (1);
}

int main (void) {
    // System Initialization
    SystemCoreClockUpdate();               // Set SystemCoreClock variable
    osKernelInitialize();                  // Initialize CMSIS-RTOS2
    osThreadNew(app_main, NULL, NULL);     // Create application main thread
    if (osKernelGetState() == osOK) {      // Kernel OK to run?
        osKernelStart();                  // Start thread execution
    }
    while(1); //will not execute unless above fails
}
```



RTX_Config.h – Kernel configuration

Edit RTX parameters to tailor the kernel to the application

- ▶ `OS_TICK_FREQ` = kernel tick frequency [Hz] (SysTick interrupts)
 - ▶ Uses **SystemCoreClock** variable to set up SysTick timer.
 - ▶ `OS_ROBIN_ENABLE` = 1 enable round-robin thread switching
= 0 disable round-robin & use timer/event scheduling
 - ▶ `OS_ROBIN_TIMEOUT` = # kernel ticks to execute before thread switch
 - ▶ `OS_THREAD_NUM` = max # active use threads (in any state)
 - ▶ `OS_THREAD_DEF_STACK_NUM` = #user thread with default stack size
 - ▶ `OS_THREAD_USER_STACK_SIZE` = total stack size (bytes) for user-provided stacks
 - ▶ `OS_STACK_SIZE` = default thread stack size if not specified
 - ▶ `OS_STACK_CHECK` = enable/disable status checking (of stack)
 - ▶ `OS_EVFLAGS_NUM` = # event flag objects
 - ▶ `OS_MUTEX_NUM` = # Mutex objects
 - ▶ `OS_SEMAPHORE_NUM` = # Semaphore objects
 - ▶ `OS_MSGQUEUE_NUM` = # Message Queue objects
 - ▶ `OS_MSGQUEUE_OBJ_MEM` = Enable allocation of Message Queue memory
 - ▶ `OS_MSGQUEUE_DATA_SIZE` = combined data storage (#bytes) for message objects
-



RTX Threads

- ▶ The scheduling unit is the **thread**
 - ▶ Threads are dynamically created, started, stopped, etc.
- ▶ Create and put each thread into the Thread List
 - ▶ Each assigned a “thread ID” to be used for scheduling, messages, flags, events, etc.

osThreadId_t *tid_ralph*; //thread ID of thread “ralph”

void *ralph* (void) { ... } //thread function “ralph”

tid_ralph = *osThreadNew*(*ralph*, *argument*, *attr*);

- ▶ *ralph* = function name
- ▶ *argument* - passed to function as start argument (default NULL)
- ▶ *attr* = thread attribute structure: priority, stack size, mem’y alloc, etc.
(NULL to use default structure values)

- ▶ *osThreadId*(); //return thread ID of current thread if not known
-



Thread attributes

- ▶ Each thread has an attribute record structure of type **osThreadAttr_t**
 - ▶ **name** - char* string thread name (ex. “MyThread”) *Thread function*
 - ▶ **cb_mem/cb_size** – memory and size of thread control block
 - ▶ **stack_mem/stack_size** – memory and size of thread stack
 - ▶ **priority** – thread priority (initially *osPriorityNormal*)
 - ▶ *several other less-used fields*
- ▶ Attribute record structure may be passed to **osThreadNew()**
 - ▶ If NULL parameter passed, default values are assigned



Thread priorities

▶ Priority levels

- ▶ `osPriorityIdle` (1) – lowest priority
- ▶ `osPriorityLow` (8)
- ▶ `osPriorityBelowNormal` (16)
- ▶ `osPriorityNormal` (24) – default priority
- ▶ `osPriorityAboveNormal` (32)
- ▶ `osPriorityHigh` (40)
- ▶ `osPriorityRealTime` (48)
- ▶ `osPriorityISR` (56) – highest priority

7 more levels for each:
Example:
 $\text{LowN (N=1..7)} = 8+N$

Likewise for all but
IDLE and ISR

▶ Thread priority set to default when thread created:

- ▶ Initial default priority `osPriorityNormal`
- ▶ If `Thread Attribute` record specified for new thread, use its priority field.

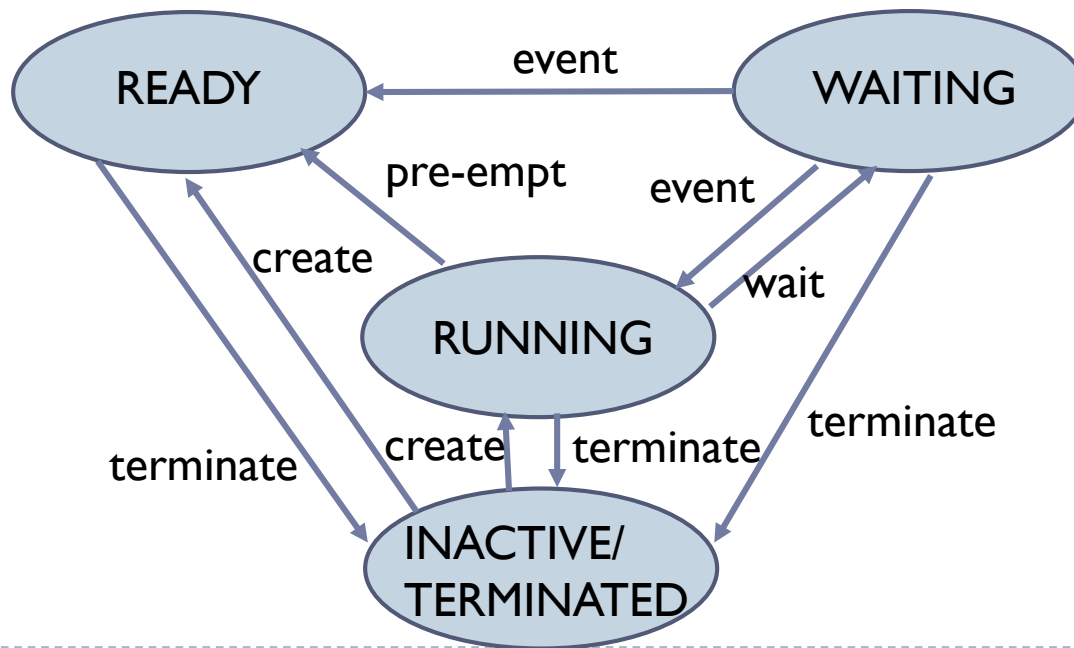
▶ Change priorities:

- ▶ `osThreadSetPriority(tid, p);` //tid = task id, new priority p
- ▶ `osThreadGetPriority();` //return current task priority

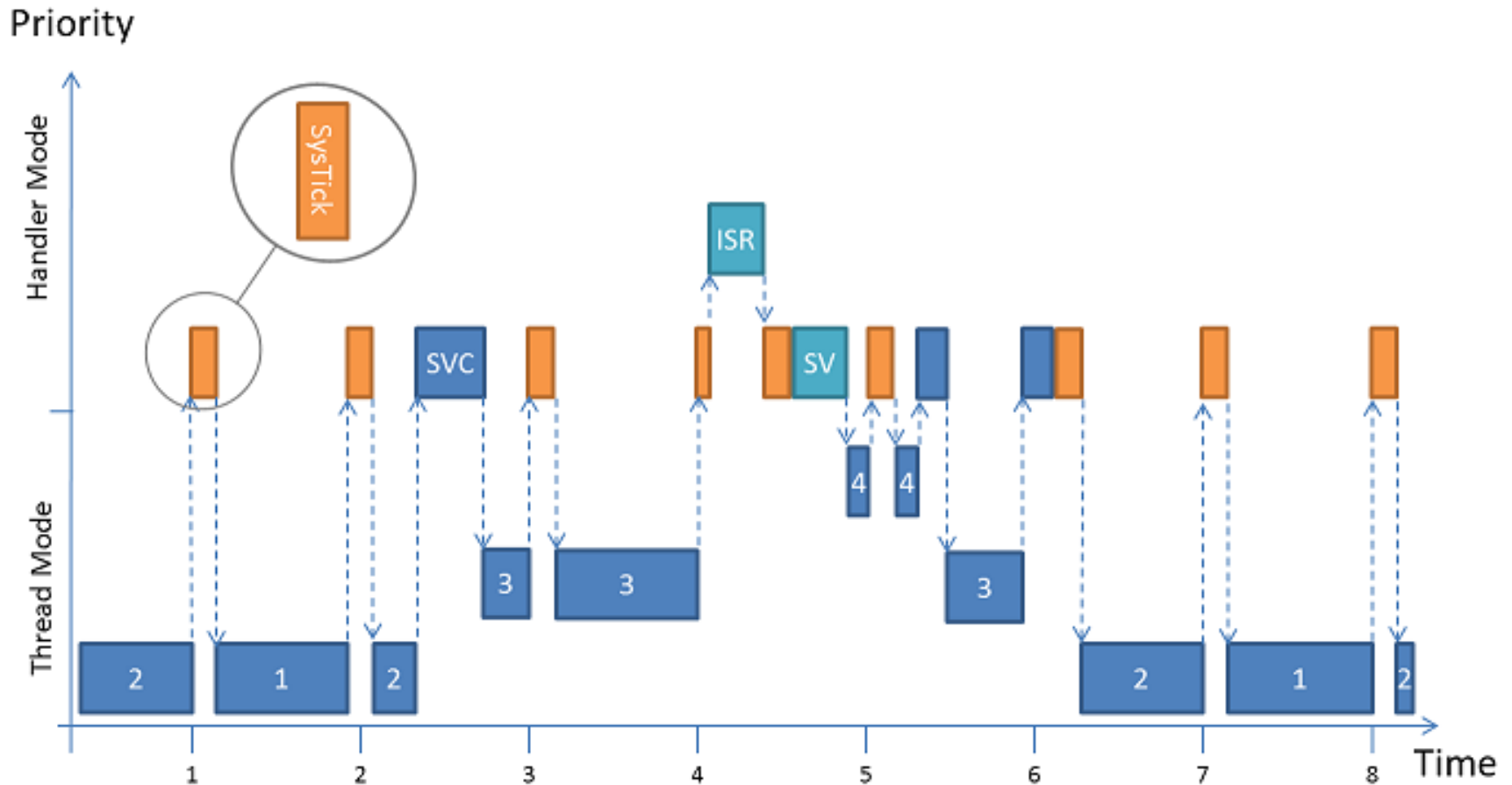


Thread states

- ▶ **RUNNING** – thread currently running
- ▶ **READY** to run, RTX chooses highest-priority
- ▶ **BLOCKED** – waiting for some event to occur
- ▶ **INACTIVE** – thread created but not active, or terminated



Thread scheduling and interrupts



Threads 1 & 2 have same priority; Thread 3 higher; Thread 4 highest

Controlling thread states

- ▶ Running thread **BLOCKED** if it must wait for an event
 - ▶ Thread flag, event signal, semaphore, mutex, message, etc.
- ▶ **osThreadYield()** – move to **READY** and pass control to next **READY** thread of same priority (continue if no ready threads)
- ▶ **osThreadSuspend(tid)** – move thread tid -> **BLOCKED**
- ▶ **osThreadResume(tid)** – move thread tid -> **READY**
- ▶ **osThreadTerminate(tid)** – move thread tid -> **INACTIVE**
 - ▶ Remove from list of active threads (terminate if running)
- ▶ **Function return values:**
 - ▶ **osOK** - successful
 - ▶ **osError** – unspecified error
 - ▶ **osErrorISR** – cannot call function from an ISR
 - ▶ **osErrorResource** – thread in wrong state for attempted action



```
/*-----  
*   Thread I 'MyThread': Sample thread  
*-----*/
```

```
#include <cmsis_os2.h>    // CMSIS RTOS2 header file
```

```
void MyThread (void * argument);           // thread function prototype  
osThreadId_t tid_MyThread;                 // thread id variable
```

```
/* Define the thread function */
```

```
void MyThread (void const *argument) {  
    while (1) {  
        ;// Insert thread code here...  
        osThreadYield(); // suspend thread  
    }  
}
```

```
/* Created before and executed after kernel initialized */
```

```
void main_thread (void) {  
    tid_MyThread = osThreadCreate (MyThread, NULL, NULL); //create thread MyThread  
    if(!tid_MyThread) return(-1); //error if thread ID is NULL  
    // .... create other new threads .....  
    osDelay(osWaitForever); //suspend this thread  
    while (1);  
}
```



/* Simple program using round-robin schedule with two threads */

#include "cmsis_os2.h" // CMSIS-RTOS2 header file

```
int counter1, counter2;  
osThreadId_t tid_job1; //thread IDs  
osThreadId_t tid_job2;
```

```
void job1 (void const *arg) { //First thread  
    while (1) { // loop forever  
        counter1++; // update the counter  
    }  
}
```

```
void job2 (void const *arg) { //Second thread  
    while (1) { // loop forever  
        counter2++; // update the counter  
    }  
}
```

```
int main (void) { //executed at startup  
    osKernelInitialize (); // set up kernel  
    tid_job1 = osThreadCreate (job1, NULL, NULL); // create and add thread "job1" to Thread List  
    tid_job2 = osThreadCreate (job2, NULL, NULL); // create and add thread "job2" to Thread List  
    osKernelStart (); // start kernel  
}
```



ARM CMSIS-RTOS scheduling policies

- ▶ Round robin schedule (OS_ROBIN_ENABLE = 1)
 - ▶ All threads assigned same priority
 - ▶ Threads allocated a fixed time
 - ▶ OS_ROBIN_TIMEOUT = ticks allocated to each thread
 - ▶ OS_TICK_FREQ = frequency (in Hz) of SysTick timer interrupts
 - ▶ Thread runs for designated time or until blocked/yield
- ▶ Round robin with preemption (OS_ROBIN_ENABLE = 1)
 - ▶ Threads assigned different priorities
 - ▶ Higher-priority thread becoming ready preempts (stops) a lower-priority running thread
- ▶ Pre-emptive (OS_ROBIN_ENABLE = 0)
 - ▶ Threads assigned different priorities
 - ▶ Thread runs until blocked, or executes osThreadYield(), or higher-priority thread becomes ready (no time limit)
- ▶ Co-operative Multi-Tasking (OS_ROBIN_ENABLE = 0)
 - ▶ All threads assigned same priority
 - ▶ Thread runs until blocked (no time limit) or executes osThreadYield();
 - ▶ Next ready thread executes



Preemptive multitasking

- ▶ `#define OS_ROBIN_ENABLE 0` in `RTX_Config.h`
- ▶ RTX suspends running thread if a higher priority thread (HPT) becomes READY
- ▶ Thread scheduler executes at system tick timer interrupt.
- ▶ Thread context switch occurs when:
 - ▶ Event set for a HPT by the running thread or by an interrupt service routine (event for which the HPT was waiting)
 - ▶ Token returned to a semaphore for which HPT is waiting
 - ▶ Mutex released for which HPT is waiting
 - ▶ Message posted to a message queue for which HPT is waiting
 - ▶ Message removed from a full message queue, with HPT waiting to send another message to that queue
 - ▶ Priority of the current thread reduced and a HPT is ready to run



Round-Robin Multitasking

- ▶ RTX gives a time slice to each thread (`OS_ROBIN_TIMEOUT`)
- ▶ Thread executes for duration of time slice, unless it voluntarily stops (via a system delay or yield function)
- ▶ RTX changes to next READY thread with same priority
 - ▶ if none – resume current thread
- ▶ Configure in *RTX_Config.h*
 - ▶ `#define OS_ROBIN_ENABLE 1`
 - ▶ `#define OS_ROBIN_TIMEOUT n`
n = #kernel ticks given to each thread (“timeout” value)



Basic wait/delay function

- ▶ Suspend a thread for a designated amount of time
- ▶ `osStatus_t osDelay (uint32_t T);`
 - ▶ Change thread state to WAITING
 - ▶ Change thread state back to READY after T kernel ticks
 - ▶ Return status = `osOK` if delay properly executed
= `osErrorISR` if `osDelay()` called from an ISR (not permitted)

```
#include "cmsis_os2.h"
```

```
void Thread_1 (void const *arg) { // Thread function
    osStatus_t status;           // capture the return status
    uint32_t delayTime;         // delay time in milliseconds
    delayTime = 1000;           // delay 1 second
    :
    status = osDelay (delayTime); // suspend thread execution
}
```

- ▶ Also available: `osStatus_t osDelayUntil (uint32_t T); //delay until time=T`



Inter-thread communication

- ▶ **Thread flag** – for thread synchronization
 - ▶ Each thread has a pre-allocated 32-bit thread flag object.
 - ▶ A thread can wait for its TFs to be set by threads/interrupts.
- ▶ **Event flag** – for thread synchronization
 - ▶ Similar to thread flags, except dynamically created
- ▶ **Semaphore** – control access to common resource
 - ▶ Semaphore object contains **tokens** (“counting” semaphore)
 - ▶ Thread can request a token (put to sleep if none available)
- ▶ **Mutex** – mutual exclusion locks
 - ▶ “lock” a resource to use it, and unlock it when done
 - ▶ Kernel suspends threads that need the resource until unlocked
- ▶ **Message Queue** (**Mail Queue** eliminated in RTOS2)
 - ▶ Queue is a first-in/first-out (FIFO) structure
 - ▶ “Message” is an integer or a pointer to a message frame
 - ▶ Suspend thread if “put” to full queue or “get” from empty queue



Thread Flags

- ▶ Thread flags not “created” – a 32-bit word with 31 thread flags; exists automatically within each thread.



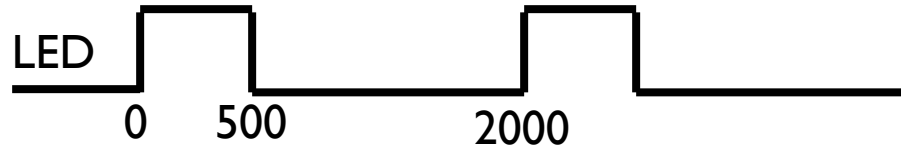
- ▶ One thread sets TFs in another thread (addressed by its thread ID)
 - ▶ **osThreadFlagsSet**(tid, flags) – set TFs of thread tid
 - ▶ flags = int32_t; each “1” bit in “flags” sets the corresponding TF
 - ▶ Example: flags=0x8002 => set/clear TF #15 and TF #1
 - ▶ **osThreadFlagsWait**(flags, option, timeout)
 - ▶ Wait for TFs corresponding to “1” bits in “flags” to be set
 - ▶ Option = osFlagsWaitAny or osFlagsWaitAll = wait for any or all of the flags
 - ▶ Timeout = 0 (check and return), osWaitForever, or time T
 - ▶ Return 32-bit value of flags (and then clear them)
 - osFlagsErrorTimeout if TFs are set before timeout T
 - osFlagsEventTimeout if no flag before timeout
 - ▶ **osThreadFlagsClear**(tid, flags) – clear TFs of thread, return current flags set
 - ▶ **osThreadFlagsGet**() – return flags currently set in this thread
-



CMSIS-RTOS thread flags example

//Thread 1

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



//Thread 2

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




```
// Thread Flag Example – Thread3 must wait for signals from both Thread1 and Thread2
#include "cmsis_os2.h"
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 both thread1 and thread2
        flags = osThreadFlagsWait(0x0003, osFlagsWaitAll, osWaitForever);
        ... //continue processing
    }
}
```



CMSIS-RTOS2 Event Flags

- ▶ Each “signal” has up to 31 “event flags” (bits 30-0 of the signal word)
- ▶ Similar to Thread Flags, but Event Flags do not “belong” to any thread
 - ▶ Wait (in BLOCKED state) for an event flag to be set
 - ▶ Set/Clear one or more event flags
- ▶ **osEventFlagsId_t** evt_id;
 evt_id = **osEventFlagsNew**(*attr) – create & initialize event flags
 - ▶ NULL argument for default values (or pointer to osEventFlagsAttr_t structure)
 - ▶ Return event flags id (evt_id)
- ▶ **osEventFlagsSet**(evt_id, flags) – set EFs in evt_id
osEventFlagsClear(evt_id, flags) – clear EFs of evt_id
 - ▶ flags = int32_t; each “1” bit in “flags” sets/clears the corresponding EF
 - ▶ Return int32_t = flags after executing the set/clear (or error code)
- ▶ **osEventFlagsWait**(evt_id, flags, options, timeout)
 - ▶ Wait for EFs corresponding to “1” bits in “flags” to be set, or until timeout
 - ▶ Options – osFlagsWaitAny or osFlagsWaitAll (any or all of the indicated flags)
 - ▶ Return current event flags or error code
 - osFlagsErrorTimeout if awaited flags not set before timeout
 - osFlagsErrorResource if evt_id not ready to be used

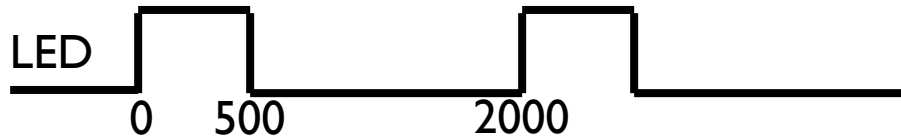


Event flags example

```
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(0);  
        osEventFlagsSet(led_flag, 0x0001); //signal ledOff thread  
        osDelay(2000);  
    }  
}
```



```
void ledOff (void constant *argument) {  
    for (;;) { // wait for signal from ledOn thread  
        osEventFlagsWait(led_flag, 0x0001, osFlagsWaitAny, osWaitForever);  
        osDelay(500);  
        LED_Off(0);  
    }  
}
```

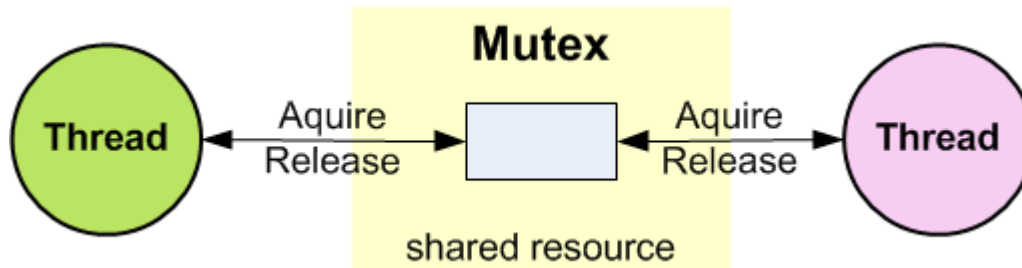


Mutual Exclusion (MUTEX)

- ▶ **Binary semaphore**
- ▶ **Provide exclusive access to a resource**
- ▶ `osMutexId_t m_id;` `//MUTEX ID`
`m_id = osMutexNew(attr);` `//create MUTEX obj`
 - ▶ `attr = osMutexAttr_t` structure or `NULL` for default
- ▶ `status = osMutexAcquire(m_id, timeout);`
 - ▶ Wait until MUTEX available or until time = “timeout”
 - ▶ `timeout = 0` to return immediately
 - ▶ `timeout = osWaitForever` for infinite wait
 - ▶ “status” = `osOK` if MUTEX acquired
 `osErrorTimeout` if not acquired within timeout
 `osErrorResource` if not acquired when timeout=0 specified
- ▶ `status = osMutexRelease(m_id);` `//release the MUTEX`
 - ▶ `status = osOK` if released, `osErrorResource` if invalid operation (not owner)

Timeout arguments
for other objects
have same options

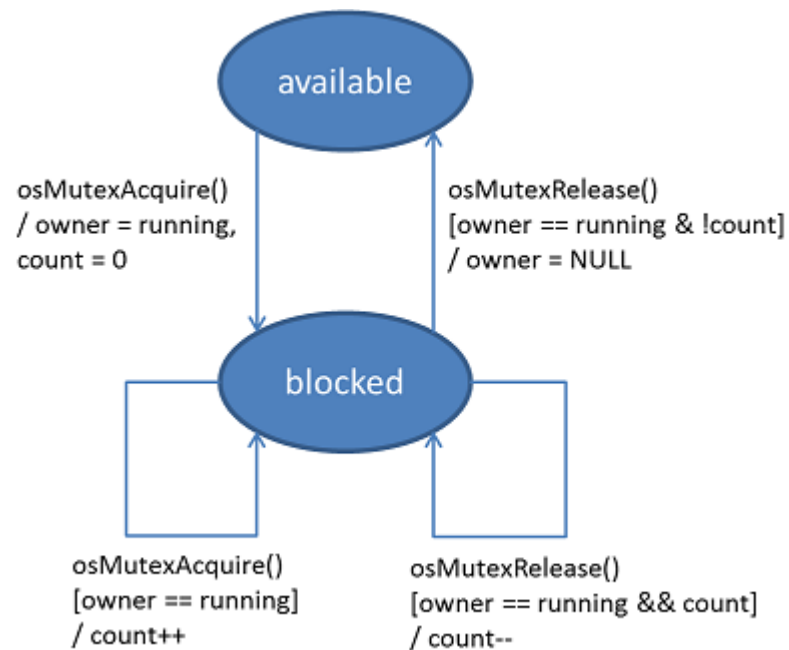




`osMutexAcquire(mutex_id, timeout)`
`osMutexRelease(mutex_id)`

Limit access to
shared resource to
one thread at a time.

Special version of a
“semaphore”



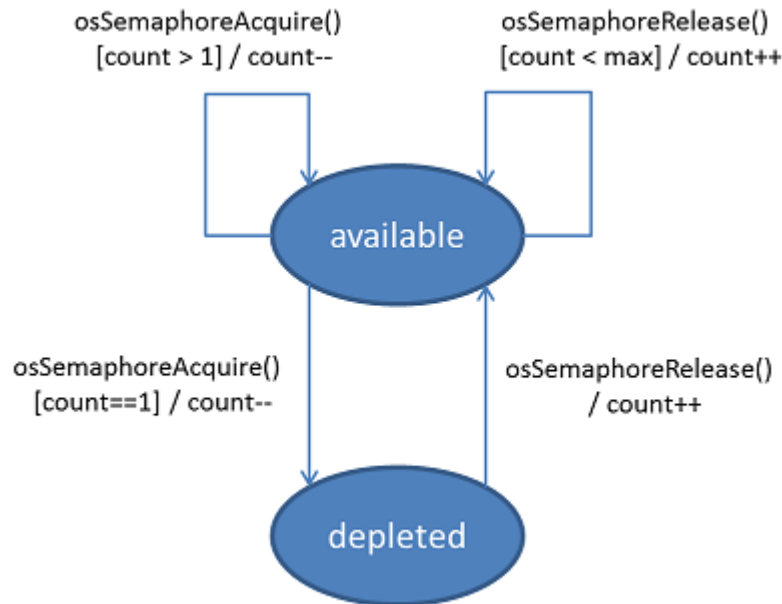
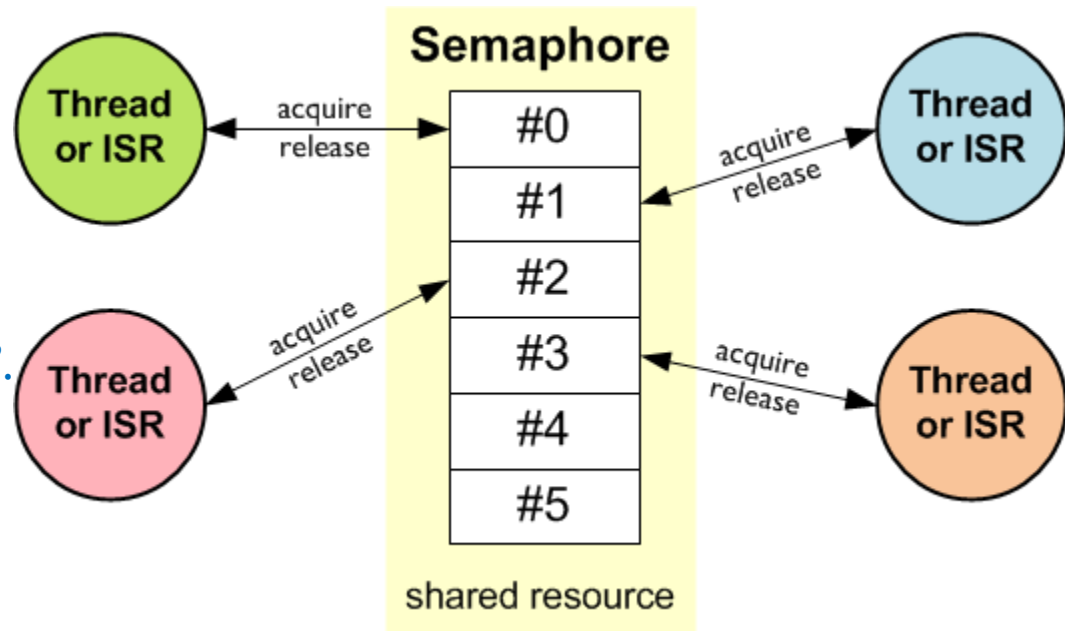
Semaphores

- ▶ **Counting Semaphore**
- ▶ **Allow up to t threads to access a resource**
- ▶ `osSemaphoreId s_id; // semaphore ID`
`s_id = osSemaphoreNew(max_tokens, init_tokens, attr);`
 - ▶ Create `s_id`; set max and initial #tokens
 - ▶ `attr` `osSemaphoreAttr_t` structure or `NULL` for defaults
- ▶ `status = osSemaphoreAcquire(s_id, timeout);`
 - ▶ Wait until token available or timeout
 - ▶ `status =`
 - `osOK` if token obtained (#tokens decremented)
 - `osErrorTimeout` if token not obtained before timeout
 - `osErrorResource` if token not obtained and timeout=0
- ▶ `status = osSemaphoreRelease(s_id);`
 - ▶ Release token
 - ▶ `status =`
 - `osOK` if token released (#tokens incremented)
 - `osErrorResource` if max token count reached
 - `osErrorParameter` if `s_id` invalid



Permit fixed number of threads/ISRs to access a pool of shared resources.

Initialize with max# of “tokens”.



`osSemaphoreAcquire(sem_id)`
`osSemaphoreRelease(sem_id)`

`osSemaphoreGetCount(sem_id)`

CMSIS-RTOS semaphore example

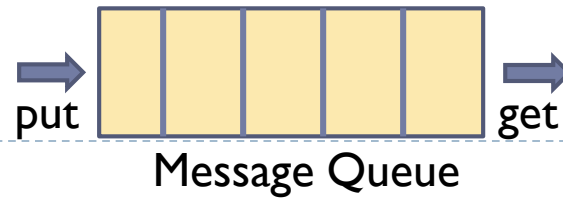
```
osSemaphoreId_t sid_Thread_Semaphore;           // semaphore id

// Main thread: Create the semaphore
sid_Thread_Semaphore = osSemaphoreNew(2, 2, NULL); //init with 2 tokens
if (!sid_Thread_Semaphore) {                     ;// Semaphore object not created, handle failure }

// Application thread: Acquire semaphore - perform task - release semaphore
osStatus_t val;
val = osSemaphoreWait (sid_Thread_Semaphore, 10); // wait up to 10 ticks
switch (val) {
    case osOK:                                     //Semaphore acquired
        // Use protected code here...
        osSemaphoreRelease (sid_Thread_Semaphore); // Return token back to a semaphore
        break;
    case osErrorTimeout:   break;                  // Not acquired within timeout
    case osErrorResource:  break;                  // Not acquired and timeout=0 ("just checking")
    default:               break;                  // Other errors
}
```

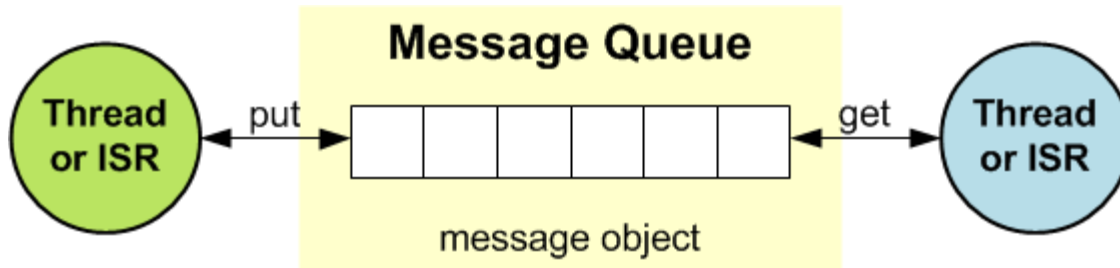


Message queues



“Message” = information to be sent

- ▶ `osMessageQueueId q_id;` // ID of queue object
- ▶ `q_id = osMessageQueueNew(msg-count, msg-size, attr);`
 - ▶ Create and initialize a message queue, return queue ID
 - ▶ Specify: max #msgs, max msg size, attributes (or NULL for defaults)
- ▶ `status = osMessageQueuePut(q_id, msg-ptr, msg-priority, timeout);`
 - ▶ Add message to queue; wait for “timeout” if queue full
 - ▶ msg-ptr = pointer to message data structure
 - ▶ Status = `osOK` : msg was put into the queue
 - = `osErrorResource` : not enough space for msg
 - = `osErrorTimeout` : no memory available at timeout
- ▶ `status = osMessageQueueGet(q_id, msg-ptr, msg-priority, timeout);`
 - ▶ Get msg from queue and put in *msg-ptr; wait for “timeout” if no message
 - ▶ Status = `osOK` : no msg available and timeout=0
 - = `osEventTimeout` : no message available before timeout
 - = `osEventMessage` : msg received (“status” is a “union” structure)



`osMessageQueuePut(mq_id, *msg_ptr, msg_prio, timeout)`
`osMessageQueueGet(mq_id, *msg_ptr, msg_prio, timeout)`

<code>osMessageQueueGetCapacity(mq_id)</code>	- max #msgs in the queue
<code>osMessageQueueGetMsgSize(mq_id)</code>	- max msg size in memory pool
<code>osMessageQueueGetCount(mq_id)</code>	- # queued msgs in the queue
<code>osMessageQueueGetSpace(mq_id)</code>	- # available slots in the queue
<code>osMessageQueueReset(mq_id)</code>	- reset to empty

`/* Message Queue creation & usage example */`

`// message object data type`

```
typedef struct {  
    uint8_t Buf[32];  
    uint8_t Idx;  
} MSGQUEUE_OBJ_t;
```

`// message queue id`

```
osMessageQueueId_t mid_MsgQ;
```

`// thread creates a message queue for 12 messages`

```
int Init_MsgQueue (void) {  
    mid_MsgQ = osMessageQueueNew(12, sizeof(MSGQUEUE_OBJ_t), NULL);  
    ....  
}
```

Continued on next slide



/* Message Queue Example Continued */

```
void Thread1 (void *argument) { // this thread sends data to Thread2
    MSGQUEUE_OBJ_t msg;
    while (1) {
        ;// Insert thread code here...
        msg.Buf[0] = 0x55; // data to send
        msg.Idx = 0; // index of data in Buf[]
        osMessageQueuePut (mid_MsgQ, &msg, 0, NULL); // send the message
        osThreadYield (); // suspend thread
    }
}

void Thread2 (void *argument) { //This thread receives data from Thread1
    MSGQUEUE_OBJ_t msg;
    osStatus_t status;
    while (1) {
        ;// Insert thread code here...
        status = osMessageQueueGet (mid_MsgQ, &msg, NULL, NULL); // wait for message
        if (status == osOK) {
            ;// process data in msg.Buf[msg.Idx]
        }
    }
}
```



Additional examples

- ▶ Examples provided in CMSIS-RTOS2 Documentation

<http://www.keil.com/pack/doc/cmsis/RTOS2/html/index.html>

- ▶ Examples provided in Keil installation directory

[C:/Keil_v5/ARM/PACK/ARM/CMSIS/5.3.0/CMSIS/RTOS2/RTX/Examples](#)

